PYTHON BASICS

The Foundation of Programming

PYTHON LOGO

python

"Simple • Powerful • Versatile"

HELLO WORLD IN PYTHON

```
# This is a simple Python program
print("Hello, World!")

# Variables and Data Types
name = "Python"          # String
version = 3.12           # Float
is_fun = True            # Boolean

# Output
print("Language:", name)
print("Version:", version)
print("Is it fun?:", is_fun)
```

OUTPUT

```
Hello, World!
Language: Python
Version: 3.12
Is it fun?: True
```

Programming Language
Python is a general-purpose programming language.

Simple Syntax
Easy to learn and write. Clean and readable code.

Interpreted
Python code is executed line by line by the interpreter.

High Level
Abstracts complex details and focuses on problem solving.

General Purpose
Used for web development, data science, AI, automation and more.

Cross Platform
Runs on Windows, macOS, Linux and many other platforms.

Open Source
Open source and community driven with huge support.

Huge Community
Large developer community and rich ecosystem of libraries.

WHY PYTHON?

- Easy to Learn and Fast
- Readable Syntax
- Large Standard Library
- Reliable and Secure
- Wide Range of Applications
- Used by Top Companies

Did You Know?

Python was created by Guido van Rossum and first released in 1991.

RUN PYTHON

Save your code in a file (e.g., main.py) and run in terminal:

```
>_ python main.py
```

python™ INSTALLATION & SETUP

Set up Python Environment in Simple Steps

Download | Install | Verify | Code | Run | Build

1. DOWNLOAD PYTHON

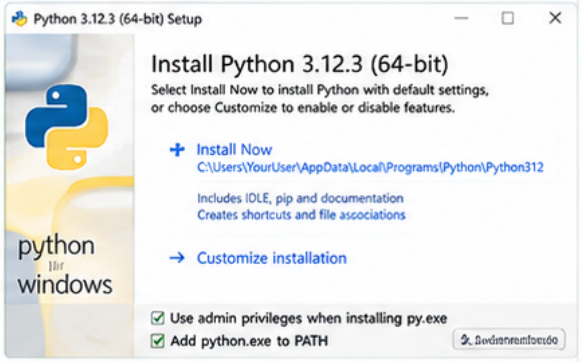
Download the latest Python version from the official website.

https://www.python.org/downloads/

Tip: Choose the latest stable version for your OS (Windows / macOS / Linux).

2. INSTALL PYTHON

- Run the installer.
- Check "Add Python to PATH".
- Choose "Install Now".
- Wait for installation to complete.
- Click Close and you're ready!


Python 3.12.3 (64-bit) Setup
Install Python 3.12.3 (64-bit)
Select Install Now to install Python with default settings, or choose Customize to enable or disable features.
Install Now
C:\Users\YourUser\AppData\Local\Programs\Python\Python312
Includes IDLE, pip and documentation
Creates shortcuts and file associations
Customize installation
python for windows
Use admin privileges when installing py.exe
Add python.exe to PATH

3. VERIFY INSTALLATION

Open Terminal / Command Prompt and run:

Check Python Version

```
> python --version
```

✓ 3.12.3

Check Pip Version

```
> pip --version
```

✓ pip 24.0

If both commands work, Python is installed correctly!

4. PYTHON DEVELOPMENT TOOLS

IDLE

Built-in Python IDE. Simple and perfect for beginners.

Start → IDLE (comes with Python)

VS Code

Powerful and lightweight code editor by Microsoft. Great for Python development.

Install Python Extension in VS Code

PyCharm

Professional IDE by JetBrains with advanced features for Python.

Community Edition (Free)

Terminal

Run Python commands, execute scripts and manage packages.

Use Command Prompt or Terminal

5. RUN YOUR FIRST PYTHON PROGRAM

Create a file: hello.py

```
1  # This is your first Python program
2  print("Hello, World!")
3
```

hello.py PY

Run in Terminal:

```
>_ python hello.py
```

Output:

```
Hello, World!
```

.py File

Python files have .extension .py

Example: script.py

6. INSTALL PACKAGES USING PIP

```
>_ pip install package_name
```

Example:

```
> pip install numpy
```

7. COMMON USEFUL COMMANDS

Command	Purpose
python --version	Check Python version
pip --version	Check Pip version
pip list	List installed packages
pip install package_name	Install a package
pip uninstall package_name	Uninstall a package

8. SETUP CHECKLIST

- ✓ Python installed
- ✓ Python added to PATH
- ✓ python --version works
- ✓ pip --version works
- ✓ Code editor installed (VS Code / PyCharm / IDLE)
- ✓ You ran your first .py file

You're Ready to Code in Python! 🐍

PRO TIP

Keep Python and Pip updated for better performance and security.

```
>_ pip install --upgrade pip
```

PYTHON SYNTAX

Write Code in a Simple and Readable Way

Easy to Learn | Readable Code | Powerful Language | Versatile Use | Open Source

1. print() – Output

print() is used to display output on the screen.

```
print("Hello, World!")
name = "Python"
print("Welcome to", name)
num = 10
print("Number:", num)
```

Output:

```
Hello, World!
Welcome to Python
Number: 10
```

2. Comments

Comments are used to explain code. Python supports single-line and multi-line comments.

```
# This is a single-line comment
print("Hello")
"""
This is a
multi-line comment
"""
print("Done")
```

Comments are ignored by Python interpreter.

3. Indentation

Indentation defines code blocks in Python. Use spaces (4 spaces recommended).

```
if True:
    print("This is indented")
    print("This is inside if block")
else:
    print("This is in else block")
```

Consistent indentation is required, otherwise you will get an error.

4. Statements

A statement is an instruction that Python can execute.

```
x = 5              # Assignment statement
y = 10             # Assignment statement
z = x + y          # Expression statement
print(z)           # Function call statement
if z > 10:
    print("z is greater than 10")
else:
    print("z is 10 or less")
```

Each line is a statement.

5. Variables

Variables are used to store data. No need to declare data type.

```
name = "Python"        # String
age = 25               # Integer
price = 19.99          # Float
is_student = True      # Boolean
data = None            # NoneType
```

Variable names are case-sensitive.

6. Keywords

Keywords are reserved words in Python and cannot be used as identifiers.

False	class	finally	is	return
None	continue	for	lambda	True
and	def	from	nonlocal	try
as	del	global	not	while
assert	elif	if	or	with
break	else	import	pass	yield

Keywords have special meaning in Python.

7. Expressions

Expressions are combinations of values, variables and operators.

```
a = 10
b = 5
sum = a + b            # 15
diff = a - b           # 5
prod = a * b           # 50
div = a / b            # 2.0
mod = a % b            # 0
exp = a ** b           # 100000
```

Expressions evaluate to a value.

8. Code Block

A block of code is a group of statements written with the same indentation.

```
for i in range(3):
    print("Iteration:", i)
    if i % 2 == 0:
        print("Even")
    else:
        print("Odd")
```

Blocks start after ':' and indentation.

Python is indentation-sensitive. Use spaces, not tabs.

Follow PEP 8 style guide for clean and readable code.

Write simple, readable and maintainable code.

Practice regularly and build your logic!

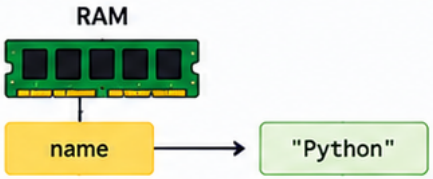
VARIABLES

Store Data for Later Use

Store Data | Reusable | Flexible | Powerful | Easy to Use

WHAT IS A VARIABLE?

A variable is a named location in memory used to store data. You can use the stored data later in your program.



You can think of a variable as a container that holds a value.

VARIABLE NAMES

- Can contain letters (a-z, A-Z), digits (0-9) and underscore (_)
- Must start with a letter or underscore
- Case-sensitive (name, Name and NAME are different)
- Cannot be a Python keyword
- Examples of invalid names: 2name, my-name, class, True

EXAMPLE

```
name = "Python"          # String
age = 25                 # Integer
price = 99.99            # Float
is_active = True         # Boolean

print(name)              # Output: Python
print(age)               # Output: 25
print(price)             # Output: 99.99
print(is_active)         # Output: True
```

OUTPUT

```
Python
25
99.99
True
```

COMMON DATA TYPES

str	int	float	bool
String	Integer	Float	Boolean
"Hello"	25	99.99	True / False

1. ASSIGNMENT

Value is assigned to a variable using the assignment operator (=).

```
x = 10
y = "Hello"
z = x + y
```

= is used to assign value

2. DYNAMIC TYPING

Python is dynamically typed. You don't need to declare the type of a variable.

```
x = 10                      # int
x = "Now I am a string"     # str
x = 3.14                    # float
x = False                   # bool
```

3. VARIABLE REASSIGNMENT

You can change the value of a variable at any time.

```
count = 1
print(count)        # 1
count = count + 1
print(count)        # 2
```

QUICK TIP
Use meaningful variable names to make your code easy to read and understand.

- Use snake_case for variable names (my_variable)
- Choose descriptive names (e.g., total_price instead of tp)
- Avoid using single letter names except in small loops
- Don't use Python keywords as variable names

python™ DATA TYPES

Different kinds of data Python can store

Built-in Types | Immutable / Mutable | Memory Efficient | Easy to Use | Powerful & Flexible

1. str (String)

str — Sequence of Unicode characters.

```
name = "Python"
message = 'Hello, World!'
multi = """Multi-line
          string"""
```

`type(name)` → `<class 'str'>`

Immutable

2. int (Integer)

int — Whole numbers, positive or negative.

```
x = 10
y = -25
z = 0
```

`type(x)` → `<class 'int'>`

Immutable

3. float (Float)

float — Decimal numbers (floating point).

```
pi = 3.14159
price = 99.99
temp = -12.5
```

`type(pi)` → `<class 'float'>`

Immutable

4. complex (Complex)

complex — Numbers with real and imaginary part.

```
z1 = 2 + 3j
z2 = -1.5 - 4.7j
z3 = complex(1, -2)
```

`type(z1)` → `<class 'complex'>`

Immutable

5. bool (Boolean)

bool — Logical values: True or False.

```
flag = True
check = False
is_valid = True
```

`type(flag)` → `<class 'bool'>`

Immutable

6. list (List)

Ordered, mutable sequence of items.

```
fruits = ["apple", "banana", "cherry"]
numbers = [1, 2, 3, 4]
mixed = [1, "hello", 3.5, True]
```

`type(fruits)` → `<class 'list'>`

Mutable

7. tuple (Tuple)

Ordered, immutable sequence of items.

```
point = (10, 20)
colors = ("red", "green", "blue")
single = (5,)   # one item tuple
```

`type(point)` → `<class 'tuple'>`

Immutable

8. set (Set)

Unordered collection of unique items.

```
unique = {1, 2, 3, 4}
colors = {"red", "green", "blue"}
mixed = {1, "a", 3, True}
```

`type(unique)` → `<class 'set'>`

Mutable

9. dict (Dictionary)

Unordered collection of key-value pairs.

```
person = {"name": "John", "age": 25}
marks = {"math": 90, "eng": 85}
empty = {}
```

`type(person)` → `<class 'dict'>`

Mutable

10. None (NoneType)

None — Represents the absence of a value.

```
a = None
result = None
data = None
```

`type(a)` → `<class 'NoneType'>`

Immutable

TYPE CHECKING

Use `type()` to check the data type.

```
type("Python")    # <class 'str'>
type(10)          # <class 'int'>
type(3.14)        # <class 'float'>
```

MUTABLE vs IMMUTABLE

Immutable (Cannot change)	Mutable (Can change)
str, int, float, complex, bool, tuple, NoneType	list, set, dict

QUICK FACTS

- Python is dynamically typed.
- You don't need to declare type.
- Types are objects in Python.
- Type can change at runtime.

EXAMPLE: TYPE IN ACTION

```
x = 10          # int
x = "Hello"     # str
x = [1, 2, 3]   # list
x = {"a": 1}    # dict
x = None        # NoneType
```

Same variable, different types at different time!

python™ TYPE CONVERSION

Convert one data type to another in Python

Flexible | Easy to Use | Safe | Essential Skill

1. int()

int

Convert value to integer.

```
x = int("10")        # 10
y = int(3.99)        # 3
z = int(True)        # 1
```

Converts string, float, bool, etc. to integer.

2. float()

float

Convert value to float.

```
x = float("10.5")    # 10.5
y = float(5)         # 5.0
z = float(True)      # 1.0
```

Converts string, int, bool, etc. to float.

3. str()

str

Convert value to string.

```
x = str(100)         # "100"
y = str(3.14)        # "3.14"
z = str(True)        # "True"
```

Converts any value to string.

4. bool()

bool

Convert value to boolean.

```
x = bool(1)          # True
y = bool("")         # False
z = bool("Hello")    # True
```

0, "", [], {}, set() → False, others → True

5. list()

Convert value to list.

```
x = list("abc")      # ['a', 'b', 'c']
y = list((1, 2, 3))  # [1, 2, 3]
z = list({4, 5})     # [4, 5]
```

Creates a new list from iterable.

6. tuple()

( )

Convert value to tuple.

```
x = tuple([1, 2, 3])   # (1, 2, 3)
y = tuple("ab")        # ('a', 'b')
z = tuple({4, 5})      # (4, 5)
```

Creates a new tuple from iterable.

7. set()

{ }

Convert value to set.

```
x = set([1, 2, 2, 3])    # {1, 2, 3}
y = set("hello")         # {'h', 'e', 'l', 'o'}
z = sel({4, 4, 5})       # {4, 5}
```

Removes duplicates, creates a set.

8. dict()

{:}

Convert value to dictionary.

```
x = dict(a=1, b=2)                  # {'a': 1, 'b': 2}
y = dict([("x", 10), ("y", 20)])    # {'x': 10, 'y': 20}
z = dict(name="Python", type="Lang") # {'name': 'Python',
                                     #  'type': 'Lang'}
```

Creates dictionary from key-value pairs or iterable.

9. type()

type()

Return the type of the value.

```
x = type(10)         # <class 'int'>
y = type(3.14)       # <class 'float'>
z = type("Python")   # <class 'str'>
a = type([1, 2, 3])  # <class 'list'>
```

Helps to check the data type of any value.

QUICK EXAMPLE

```
a = "123"          # '123' is string
b = int(a)         # 123 (int)
c = float(a)       # 123.0 (float)
d = str(45)        # '45' (string)
e = list("hi")     # ['h', 'i'] (list)
```

IMPORTANT NOTES

- Always check the input before converting.
- Invalid conversion will raise ValueError.
- Use type() to know the current data type.
- Type conversion does not change the original value.

CONVERSION FLOW

PRO TIP

Understand data types well to write bug-free and efficient Python code.

STRINGS

Work with Text Data in Python

Text Based | Immutable | Sequence | Powerful | Easy to Use

1. STRING (Overview)

String

String is a sequence of characters enclosed in quotes.

```
type("Hello")   → <class 'str'>
type('Python')  → <class 'str'>
```

Strings are immutable.

2. QUOTES

Single Quotes

```
'Hello, World!'
```

Double Quotes

```
"Hello, World!"
```

Triple Quotes (Multi-line)

```
"""This is
  a multi-line
  string"""
```

Single and double quotes are same.
Use triple quotes for multi-line strings.

3. INDEXING

Access characters using index (starts from 0)

```
text = "Python"

       0   1   2   3   5
       P   y   t   h   n
```

```
text[0]          # 'P'
text[1]          # 'y'
text[-1]         # 'n'
text[-2]         # 'o'
```

Negative index starts from end (-1).

4. SLICING

Extract a part of string using [start:end]

```
text = "Python Programming"
 0 1  2  3  4 5 6 7 8 9 10 11 12 13 14
 P y  t  h  o n P r o g r  a  m  i  n  g
```

```
text[0:6]        # 'Python'
text[7:18]       # 'Programming'
text[:6]         # 'Python'
text[7:]         # 'Programming'
text[::2]        # 'Pto rgamn'
```

Format: [start:end:step] (step is optional)

5. f-STRINGS (Formatted Strings)

Embed variables or expressions in strings.

```
name = "John"
age = 25
score = 95.5

message = f"My name is {name}."
info = f"Age: {age}, Score: {score}"
```

f-string starts with 'f' before quotes.
Very useful for formatting.

6. ESCAPE CHARACTERS

Use backslash (\) to escape special characters.

Escape	Meaning	Example
\n	New Line	"Hello\nWorld"
\t	Tab	"A\tB"
\'	Single Quote	'It\'s ok'
\"	Double Quote	"He said \"Hi\""
\\	Backslash	"C:\\Users\\John"
\r	Carriage Return	"Hello\rWorld"

Escape characters start with backslash (\).

7. STRING OPERATIONS

Common operations with strings.

```
text = "Python"
text + " Programming"   # Concatenation
text * 3                # Repetition
"Py" in text            # Membership
"Py" not in text        # Not Membership
len(text)               # Length
```

Strings support many useful operations.
Check length with len().

8. COMMON METHODS (Examples)

Some frequently used string methods.

```
text = "  Python Programming  "
text.lower()          # python programming
text.upper()          # PYTHON PROGRAMMING
text.strip()          # 'Python Programming'
text.replace("Python", "Java')   # Replace
text.split()          # ['Python', 'Programming']
```

Explore more methods: dir(str)
Many powerful methods are available.

QUICK TIPS

- Strings are immutable (cannot be changed).
- Use quotes to create strings.
- Index starts from 0, use negative index from end.
- Use f-strings for clean and readable code.

STRING METHODS (COMMON)

.lower()	.upper()	.strip()	.replace(a,b)	.split(sep)	.join(iterable)
Convert to lowercase	Convert to uppercase	Remove leading and trailing spaces	Replace all occurrences of a with b	Split string by separator (default: space)	Join elements of iterable with string

STRING METHODS

Built-in Methods to Work with Strings in Python

Built-in Functions | Powerful & Fast | Easy to Use | Clean & Readable

OVERVIEW

String methods are functions associated with strings. They help to manipulate and work with text data easily.

SYNTAX

```
string.method(arguments)
```

EXAMPLE

```
text = "  Python Programming  "
text.upper()        # 'PYTHON PROGRAMMING'
text.strip()        # 'Python Programming'
text.replace("Python", 'Java')
                    # '  Java Programming  '
text.split()    # ['Python', 'Programming']
"-".join(['A','B','C'])    # 'A-B-C'
```

All string methods return a new string. They do not change the original string.

METHOD	DESCRIPTION	SYNTAX	EXAMPLE	RESULT
upper()	Converts all characters to uppercase.	`string.upper()`	`"Python"`	`'PYTHON'`
lower()	Converts all characters to lowercase.	`string.lower()`	`"PYTHON"`	`'python'`
strip()	Removes leading and trailing whitespace.	`string.strip()`	`"  Python  "`	`'Python'`
replace()	Replaces old substring with new substring.	`string.replace(old, new)`	`"one two one"`	`'one two three'`
split()	Splits string into a list by separator.	`string.split(sep)`	`"a,b,c"`	`['a', 'b', 'c']`
join()	Joins elements of an iterable into a string.	`sep.join(iterable)`	`"-".join(['a','b','c'])`	`'a-b-c'`
find()	Finds first occurrence of substring. Returns index.	`string.find(sub)`	`"Hello World"`	`6`
startswith()	Checks if string starts with given prefix.	`string.startswith(prefix)`	`"Python Code"`	`True`
endswith()	Checks if string ends with given suffix.	`string.endswith(suffix)`	`"Code.py"`	`True`

MORE EXAMPLES

```
"hello".upper()              # 'HELLO'
"HELLO".lower()              # 'hello'
"  hello  ".strip()          # 'hello'
"I love Python".replace("Python", "Java")
                             # 'I love Java'
"a|b|c".split("|")           # ['a','b','c']
"/".join(['2025','08','19'])
                             # '2025/08/19'
"Hello World".find("World")  # 6
"Python Code".startswith("Python")
                             # True
"file.txt".endswith(".txt")  # True
```

CHAIN METHODS

You can chain methods together.

```
text = "  python programming  "
result = text.strip().title().replace("Python","Python")
print(result)    # 'Python Programming'
```

strip() → title() → replace()

Methods can be combined to perform powerful text operations.

QUICK TIPS

- Strings are immutable.
- Methods return new strings.
- Use `help(str)` to see all methods.
- Check for None before calling methods.

IMMUTABLE NATURE

String methods do not change the original string.

```
text = "Python"
new_text = text.upper()
print(text)      # Python (unchanged)
print(new_text)  # PYTHON (new string)
```

COMMON STRING METHODS (QUICK LIST)

upper() | lower() | strip() | replace() | split()

join() | find() | startswith() | endswith()

PRO TIP

Practice these methods daily to become a Python string expert!

python™ STRING FORMATTING

Different Ways to Format Strings in Python

Easy to Read | Powerful | Flexible | Very Useful

WHAT IS STRING FORMATTING?

String formatting is used to insert variables or values inside a string in a clean and readable way.

- **F-String** — Modern and easy to use (Python 3.6+)
- **Placeholders** — Reserved spots for values inside string
- **Variables** — Values stored in variables can be inserted
- **Formatting** — Control number, text and date formats
- **Alignment** — Align text left, right or center

Formatting makes your output clean, professional and easier to read.

1. F-STRING (Recommended)

F-strings are the modern and easiest way. Use f before the string.

```
name = "John"
age = 25
price = 99.99

msg = f"Hello {name}, you are {age} years old."
msg2 = f"The price is ${price:.2f}."
```

OUTPUT

```
Hello John, you are 25 years old.
The price is $99.99.
```

FEATURES

- Fast and readable
- Support expressions: {age + 5}
- Format numbers: {price:.2f}

2. format() METHOD

Use the str.format() method with placeholders inside curly braces {}.

```
name = "John"
age = 25
price = 99.99

msg = "Hello {}, you are {} years old.".format(name, age)
msg2 = "The price is ${:.2f}.".format(price)
```

OUTPUT

```
Hello John, you are 25 years old.
The price is $99.99.
```

FEATURES

- Older but very powerful
- You can reuse values: {0}, {1}, ...
- Can specify index: {1}, {0} etc.

3. % (PERCENT) OPERATOR

Old style formatting using % operator.

```
name = "John"
age = 25
price = 99.99

msg = "Hello %s, you are %d years old." % (name, age)
msg2 = "The price is $%.2f." % price
```

OUTPUT

```
Hello John, you are 25 years old.
The price is $99.99.
```

FEATURES

- Older style (Python 2 try also)
- Uses %s (string), %d (int), %f (float)
- Still useful in some cases

4. PLACEHOLDERS EXPLAINED

Placeholder	Meaning	Example	Result
{}	Default order	"Hello {}, {}".format("A", "B")	Hello A, B
{0}, {1}	Index order	"{1} is {0}".format("first", "second")	second is first
{name}	Named placeholder	"Hello {name}".format(name="John")	Hello John
{:.2f}	Float with 2 decimals	"{:.2f}".format(3.14159)	3.14
{:,}	Comma separator	"{:,}".format(1234567)	1,234,567

5. ALIGNMENT (In format())

Syntax	Meaning	Example	Result
{:<10}	Left Align	"{:<10}".format("Left")	Left
{:>10}	Right Align	"{:>10}".format("Right")	Right
{:^10}	Center Align	"{:^10}".format("Center")	Center

10 is the total width. You can change it as per your need.

6. FORMAT SPECIFIERS (Common)

%s / {} → String
%d / {} → Integer
%f / {} → Float
%.2f / {:.2f} → Float (2 decimal)
%x / {:x} → Hexadecimal
%o / {:o} → Octal

QUICK TIPS

- Use f-strings for clean and modern code.
- Use .format() when you need more control.
- Use % operator for older code or simple cases.

Real World Example

```
1  name = "Alice"
2  age = 30
3  message = f"Hello {name}, next year you will be {age + 1}."
4  print(message)
```

→

Output:

```
Hello Alice, next year you will be 31.
```

NUMBERS & MATH

Perform Mathematical Operations in Python

Easy to Use | Fast Execution | Accurate Results | Built-in Functions

NUMBER TYPES

Python has 3 main numeric types.

int — Integer
Whole numbers (... -2, -1, 0, 1, 2 ...)

float — Float
Decimal numbers (... -1.5, 0.0, 2.5 ...)

complex — Complex
Numbers with real and imaginary part (a + bj)

BASIC EXAMPLE

```
a = 10          # int
b = 3.14        # float
c = 2 + 3j      # complex
```

OUTPUT

```
a -> 10
b -> 3.14
c -> (2+3j)
```

ARITHMETIC OPERATORS

Operator	Name	Description	Example	Result
+	Addition	Adds two numbers	10 + 5	15
-	Subtraction	Subtracts right number from left	10 - 5	5
*	Multiplication	Multiplies two numbers	10 * 5	50
//	Floor Division	Divides and returns integer part	10 // 3	3
%	Modulus	Returns the remainder of division	10 % 3	1
**	Exponentiation	Returns power of a number	2 ** 3	8

IMPORTANT NOTES

- // always returns integer part (floored result).
- % returns remainder and has same sign as divisor.
- ** means "to the power of".

```
a = 17
b = 5
print(a // b)   # 3
print(a % b)    # 2
print(2 ** 4)   # 16
```

MATH FUNCTIONS

abs()	Returns absolute value	abs(-10) = 10
round()	Rounds to nearest integer	round(3.67) = 4
pow()	Returns x raised to the power y	pow(2, 3) = 8

EXAMPLES

```
abs(-25)                # 25
round(3.14159, 2)       # 3.14
round(2.5)              # 2
pow(2, 5)               # 32
pow(3, 3)               # 27
```

QUICK TIPS

- Use // when you want whole number result.
- Use % to check even/odd: n % 2 == 0 (Even)
- Use ** for calculations like square, cube, etc.
- Use round() for clean decimal output.

PRO TIP

Combine operators and functions to solve powerful math problems!

```
result = round(((2**3 + abs(-5)) / 3), 2)
print(result)   # 4.33
```

REAL WORLD USE

- Calculating discounts and percentages
- Game score and level calculations
- Scientific and statistical calculations
- Data analysis and machine learning

COMPARISON & LOGICAL OPERATORS

Compare Values & Combine Conditions in Python

Compare Values | Make Smart Decisions | Build Logic | Essential for Programs

1. COMPARISON OPERATORS

Comparison operators are used to compare two values. They always return True or False.

Operator	Name	Description	Example	Result
==	Equal to	Checks if values are equal	5 == 5	True
!=	Not equal to	Checks if values are not equal	5 != 3	True
>	Greater than	Checks if left value is greater than right value	7 > 3	True
<	Less than	Checks if left value is less than right value	2 < 5	True
>=	Greater than or equal to	Checks if left value is greater than or equal to right value	5 >= 5	True
<=	Less than or equal to	Checks if left value is less than or equal to right value	3 <= 7	True

EXAMPLES

```
a = 10
b = 20
print(a == 10)        # True
print(a != b)         # True
print(a > b)          # False
print(a < b)          # True
print(a >= 10)        # True
print(b <= 20)        # True
```

NOTE

- Result of comparison is always either True or False.
- You can compare numbers, strings, lists, etc.
- Comparison is case sensitive for strings.

2. LOGICAL OPERATORS

Logical operators are used to combine multiple conditions.

Operator	Name	Description	Example	Result
and	AND	Returns True if both conditions are True	(5 > 3) and (2 < 4)	True
or	OR	Returns True if at least one condition is True	(5 > 3) or (2 > 4)	True
not	NOT	Reverses the result. Returns True if condition is False	not(5 > 3)	False

EXAMPLES

```
x = 10
y = 20
z = 30
print((x < y) and (y < z))      # True
print((x > y) or (y < z))       # True
print(not(x > y))               # True
print(not(x < y))               # False
print((x < y) and (z < y))      # False
print((x > y) or (z > y))       # True
```

TRUTH TABLE

A	B	A and B	A or B	not A
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

QUICK TIPS

- Comparison operators help in decision making.
- Logical operators help in combining multiple conditions.
- Use parentheses to make your conditions clear.

PRO TIP

You can combine comparison and logical operators together!

```
(x < y) and (y < z) or (x == 10)  # True
```

REAL WORLD USE

- User login validation
- Form data verification
- Game conditions
- Data filtering and searching

IMPORTANT

- == is for comparison, not assignment.
- and has higher precedence than or.
- Use not to reverse any condition.

python™ CONDITIONAL STATEMENTS

Make Decisions in Your Code

Decision Making | Control Flow | Smart Logic | Essential for Programs

1. OVERVIEW

Conditional statements allow your program to make decisions and execute different blocks of code based on conditions.

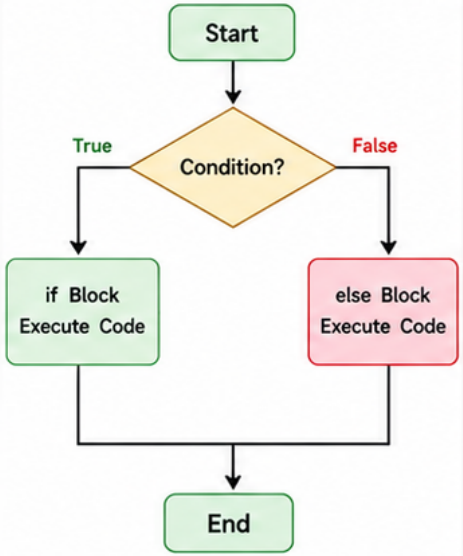
Python supports if, elif, and else to handle conditions.

2. SYNTAX

```
if condition:
    # code block
elif another_condition:
    # code block
else:
    # code block
```

Indentation is very important in Python. Use 4 spaces for each block.

3. FLOW DIAGRAM



4. EXAMPLES

Example 1: if

```
x = 10
if x > 5:
    print("x is greater than 5")
```

Output:
x is greater than 5

Example 2: if-else

```
x = 3
if x % 2 == 0:
    print("Even number")
else:
    print("Odd number")
```

Output:
Odd number

Example 3: if-elif-else

```
marks = 85
if marks >= 90:
    print("Grade: A")
elif marks >= 75:
    print("Grade: B")
else:
    print("Grade: C")
```

Output:
Grade: B

5. CONDITIONS & COMPARISON OPERATORS)

Conditions return True or False.

Operator	Meaning	Example	Result
==	Equal to	5 == 5	True
!=	Not equal to	5 != 3	True
>	Greater than	7 > 3	True
<	Less than	2 < 5	True
>=	Greater than or equal to	5 >= 5	True
<=	Less than or equal to	3 <= 7	True

6. LOGICAL OPERATORS

Combine multiple conditions.

Operator	Meaning	Example	Result
and	True if both are True	(5 > 3) and (2 < 4)	True
or	True if any one is True	(5 > 3) or (2 > 4)	True
not	Reverses the result	not (5 > 3)	False

7. NESTED IF (Decision inside Decision)

```
x = 15
if x > 0:
    if x % 2 == 0:
        print("Positive Even")
    else:
        print("Positive Odd")
else:
    print("Negative Number")
```

Output:
Positive Odd

The inner if is executed only if the outer if condition is True.

8. REAL WORLD EXAMPLE

Simple Login System

```
username = "admin"
password = "1234""

user = input("Enter username: ")
pwd  = input("Enter password: ")

if user == username and pwd == password:
    print("Login Successful ✅")
else:
    print("Invalid Credentials ❌")
```

Output (Example)

```
Enter username: admin
Enter password: 1234
Login Successful ✅
```

9. QUICK REFERENCE

- Use if to check a condition.
- Use elif for multiple conditions.
- Use else for the default case.
- Combine conditions using and, or, not.
- Use comparison operators to build conditions.
- Indentation is must (4 spaces).

QUICK TIPS

- Conditions must return True or False.
- Use meaningful variable names.
- Test all possible conditions.

BEST PRACTICES

- Keep conditions simple and readable.
- Avoid too many nested if statements.
- Use comments to explain complex logic.

PRO TIP

Good use of conditional statements makes your code smarter, faster, and bug-free!

LOOPS IN PYTHON

Repeat Actions Efficiently

Repeat Actions | Improve Productivity | Save Time & Code | Essential for Programs

1. LOOP OVERVIEW

Loops are used to repeat a block of code multiple times.

- for loop → When you know the number of times.
- while loop → When you don't know the exact number.

Loops help to automate repetitive tasks and make your code clean and efficient.

2. LOOP KEYWORDS

Keyword	Meaning
for	Iterates over a sequence (list, tuple, string, range, etc.)
while	Repeats as long as condition is True
range()	Generates a sequence of numbers
break	Terminates the loop completely
continue	Skips the current iteration and continues to next
pass	Does nothing, used as a placeholder

Indentation is very important in Python. Use 4 spaces for each block.

3. FOR LOOP

Used when the number of iterations is known.

```
# Iterate over a list
fruits = ["apple", "banana", "mango"]
for fruit in fruits:
    print(fruit)
```

Using range()

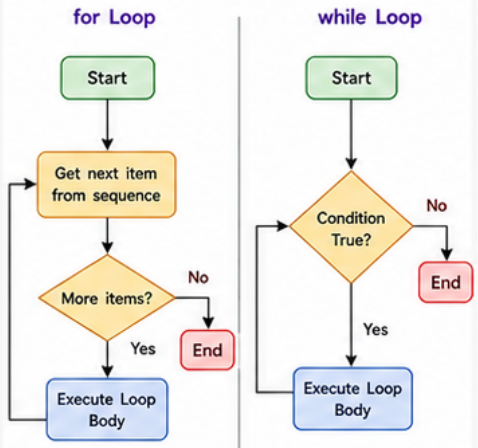
```
for i in range(5):  # 0 to 4
    print(i, end=" ")

# Output: 0 1 2 3 4
```

4. WHILE LOOP

Used when the number of iterations is unknown.

```
count = 1
while count <= 5:
    print(count)
    count += 1
# Output:
# 1
# 2
# 3
# 4
# 5
```

5. LOOP FLOW DIAGRAMS



6. LOOP CONTROL STATEMENTS

break

Terminates the loop completely.

```
for i in range(5):
    if i == 3:
        break      # stops when i == 3
    print(i)       # Output: 0 1 2
```

continue

Skips the current iteration.

```
for i in range(5):
    if i == 2:
        continue  # skip 2
    print(i)      # Output: 0 1 3 4
```

pass

Does nothing, used as a placeholder.

```
for i in range(3):
    pass        # placeholder for future code
```

7. NESTED LOOP

A loop inside another loop.

```
for i in range(1, 4):
    for j in range(1, 4):
        print(f"{i} x {j} = {i*j}")
```

Output:

```
1 x 1 = 1
1 x 2 = 2
1 x 3 = 3
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
```

8. ITERATION CONCEPT

Each repetition of a loop is called an iteration.

- First iteration → first time execution of loop body
- Second iteration → second time execution
- ...
- N-th iteration → loop runs N times

9. COMMON range() USES

Syntax	Description	Output
range(5)	0 to 4	0, 1, 2, 3, 4
range(1, 6)	1 to 5	1, 2, 3, 4, 5
range(1, 10, 2)	1 to 9, step 2	1, 3, 5, 7, 9
range(10, 0, -2)	10 to 2, step -2	10, 8, 6, 4, 2

QUICK TIPS

- Use for loop when number of times is known.
- Use while loop when number of times is unknown.
- Use break to exit early.
- Use continue to skip unwanted iterations.
- Use pass as a placeholder.

PRO TIP

Infinite loop example (Be careful!)

```
while True:
    print("Running...")
    break   # remove break → infinite loop!
```

REAL WORLD USE CASES

- Reading files line by line
- Processing items in a list
- Retry mechanisms
- Game loops
- Data processing & automation

REMEMBER

Loops reduce repetition in code and make it shorter, cleaner and easier to maintain.

LISTS IN PYTHON

Ordered, Mutable, and Versatile

Ordered Collection | Mutable (Changeable) | Heterogeneous Elements | Most Used Data Type

1. LIST OVERVIEW

A list is an ordered collection of items.

- Mutable (changeable)
- Can store different data types
- Allows duplicate values
- Indexed (starts from 0)

```
my_list = [10, "hello", 3.14, True]
print(my_list)
# Output: [10, 'hello', 3.14, True]
```

2. CREATE LISTS

- Using square brackets []

```
numbers = [1, 2, 3, 4, 5]
```

- Using list() constructor

```
fruits = list(("apple", "banana", "mango"))
```

- Empty list

```
empty_list = []
```

3. ACCESSING LIST ELEMENTS

Indexing (starts from 0)

```
lst = ["a", "b", "c", "d", "e"]
print(lst[0])      # a
print(lst[2])      # c
print(lst[-1])     # e (last item)
```

Slicing

```
lst = [0, 1, 2, 3, 4, 5]
print(lst[1:4])      # [1, 2, 3]
print(lst[:3])       # [0, 1, 2]
print(lst[3:])       # [3, 4, 5]
print(lst[::2])      # [0, 2, 4]
```

List Info

```
lst = [10, 20, 30, 40]
print(len(lst))      # 4
print(min(lst))      # 10
print(max(lst))      # 40
```

Membership

```
lst = [10, 20, 30]
print(20 in lst)       # True
print(50 not in lst)   # True
```

4. BASIC OPERATIONS

Concatenation

```
a = [1, 2]
b = [3, 4]
print(a + b)
# [1, 2, 3, 4]
```

Repetition

```
a = [1, 2]
print(a * 3)
# [1, 2,1, 2, 1, 2]
```

Iteration

```
lst = [10, 20, 30]
for item in lst:
    print(item)
# 10
# 20
# 30
```

List Comprehension

```
squares = [x*x
for x in range(5)]
print(squares)
# [0, 1, 4, 9, 16]
```

5. COMMON LIST METHODS

Method	Description	Example	Result
append()	Adds an item at the end	`lst = [1, 2]` `lst.append(3)`	`[1, 2, 3]`
extend()	Extends list by adding items from another list	`lst = [1, 2]` `lst.extend([3, 4])`	`[1, 2, 3, 4]`
insert()	Inserts item at given position	`lst = [1, 3]` `lst.insert(1, 2)`	`[1, 2, 3]`
remove()	Removes first occurrence of item	`lst = [1, 2, 3]` `lst.remove(2)`	`[1, 3]`
pop()	Removes and returns item at index (default last item)	`lst = [1, 2, 3]` `x = lst.pop(1)`	`x = 2` `[1, 3]`
clear()	Removes all items from list	`lst = [1, 2, 3]` `lst.clear()`	`[]`
index()	Returns index of first occurrence of item	`lst = [10, 20, 30]` `lst.index(20)`	`1`
count()	Returns number of times item occurs in list	`lst = [1, 2, 2, 3, 2]` `lst.count(2)`	`3`

6. NESTED LIST

```
matrix = [[1, 2], [3, 4], [5, 6]]
print(matrix[0])      # [1, 2]
print(matrix[1][0])   # 3
```

7. COPY LIST

```
lst = [1, 2, 3]
copy1 = lst.copy()      # using copy()
copy2 = lst[:]          # using slicing
copy3 = list(lst)       # using list()
```

8. LIST METHODS QUICK REFERENCE

- Use append() to add single item at end.
- Use extend() to add multiple items.
- Use insert() to add item at specific position.
- Use remove() to delete by value.
- Use pop() to delete by index and get value.
- Use clear() to empty the list.
- Use index() to find position of item.
- Use count() to count occurrences.

9. EXAMPLES

Building a List

```
lst = []
lst.append(10)
lst.append(20)
lst.append(30)
print(lst)
# [10, 20, 30]
```

Remove Duplicates

```
lst = [1, 2, 2, 3, 3, 4]
unique = list(set(lst))
print(unique)
# [1, 2, 3, 4]
```

Reverse List

```
lst = [1, 2, 3, 4]
lst.reverse()
print(lst)
# [4, 3, 2, 1]
```

QUICK TIPS

- Lists are mutable (changeable).
- Indexing starts from 0.
- Negative index starts from the end.

PRO TIP

Use list comprehensions for clean and concise code.

```
squares = [x*x for x in range(10)]
```

REAL WORLD USES

- Storing collections of data
- Task lists, shopping carts
- Data processing & analysis
- Game development (scores, levels)

NOTE

- Lists can grow or shrink in size.
- Very flexible and widely used in Python.

LIST SLICING & SORTING

Work Smarter with Python Lists

Manipulate Lists Easily | Powerful Methods | Save Time & Code | Better Code Quality

1. LIST SLICING BASICS

Slicing returns a part of the list.

```
lst   = [10, 20, 30, 40, 50, 60]
Index:   0  1   1   2   3   4   5
```

Syntax	Description	Example	Result
lst[start:end]	Items from start to end-1	lst[1:4]	[20, 30, 40]
lst[:end]	From beginning to end-1	lst[:3]	[10, 20, 30]
lst[start:]	From start to end	lst[2:]	[30, 40, 50, 60]
lst[:]	Copy of the whole list	lst[:]	[10, 20, 30, 40, 50, 60]
lst[-n:]	Last n items	lst[-2:]	[50, 60]
lst[:-n]	All except last n items	lst[:-2]	[10, 20, 30, 40]
lst[::step]	Step / stride	lst[::2]	[10, 30, 50]
lst[::-1]	Reverse using slicing	lst[::-1]	[60, 50, 40, 30, 20, 10]

2. LIST SLICING VISUAL GUIDE

```
lst = [10, 20, 30, 40, 50, 60]
Index:  0   1   3   4   5
```

3. COMMON LIST METHODS

Method	Description	Example	Result
reverse()	Reverses the list in place	lst.reverse()	lst = [60, 50, 40, 30, 20, 10]
sort()	Sorts the list in ascending order	lst.sort()	lst = [10, 20, 30, 40, 50, 60]
sorted()	Returns a new sorted list	new_lst = sorted(lst)	new_lst = [10, 20, 30, 40, 50, 60]
copy()	Returns a shallow copy of the list	lst2 = lst.copy()	lst2 = [10, 20, 30, 40, 50, 60]
len()	Returns number of items	len(lst)	6
min()	Returns smallest item	min(lst)	10
max()	Returns largest item	max(lst)	60

4. SORTING OPTIONS

Ascending (Default)

```
lst = [40, 10, 30, 20]
lst.sort()
# [10, 20, 30, 40]
```

Descending Order

```
lst.sort(reverse=True)
# [40, 30, 20, 10]
```

Sort with Key (Example: by Length)

```
words = ["apple", "kiwi", "banana", "fig"]
words.sort(key=len)
# ['fig', 'kiwi', 'apple', 'banana']
```

5. EXAMPLES

```
# Original List
lst = [5, 3, 9, 1, 7]
print("Original :", lst)
# Original : [5, 3, 9, 1, 7]
# Slicing Examples
print("lst[1:4]  :", lst[1:4])      # [3, 9, 1]
print("lst[:3]   :", lst[:3])       # [5, 3, 9]
print("lst[2:]   :", lst[2:])       # [9, 1, 7]
print("lst[::2]  :", lst[::2])      # [5, 9, 7]
print("lst[::-1] :", lst[::-1])     # [7, 1, 9, 3, 5]
# Sorting & Methods
lst.sort()
print("Sorted      :", lst))        # [1, 3, 5, 7, 9]
lst.reverse()
print("Reversed    :", lst)         # [9, 7, 5, 3, 1]
print("Min       :", min(lst))      # 1
print("Max       :", max(lst))      # 9
print("Length    :", len(lst))      # 5
print("Copy      :", lst.copy())    # [9, 7, 5, 3, 1]
```

6. QUICK REFERENCE

Slicing Quick Reference

- lst[start:end] → from start to end-1
- lst[:end] → from beginning
- lst[start:] → from start to end
- lst[::step] → step / stride
- lst[::-1] → reverse list

Method Quick Reference

- reverse() → reverse in place
- sort() → sort in ascending
- sort(reverse=True) → sort descending
- sorted() → returns new sorted list
- copy() → return a copy
- len() → number of items
- min() → smallest item
- max() → largest item

QUICK TIPS

- Use slicing to work with parts of the list.
- sorted() keeps original list unchanged.
- Use reverse=True in sort() for descending.
- copy() is useful to avoid changes in original list.

PRO TIP

Combine slicing with steps for powerful data manipulation.

```
lst[::2]    # every 2nd item
lst[::-1]   # reverse list
```

REAL WORLD USE CASES

- Data analysis & cleaning
- Sorting records (A to Z, Z to A)
- Getting top/bottom values
- Pagination using slicing

NOTE

Lists are mutable (changeable). Methods like sort(), reverse() change the original list.

TUPLES IN PYTHON

Immutable, Ordered, and Reliable

Immutable Data | Ordered Collection | Faster & Memory Efficient | Used as Dictionary Keys

1. TUPLE OVERVIEW

A tuple is an ordered collection of items.

- Immutable (cannot be changed)
- Allows duplicate values
- Ordered (maintains index)
- Can store different data types

```
t = (10, "hello", 3.14, True)
print(t)
# Output: (10, 'hello', 3.14, True)
```

2. IMMUTABLE (CANNOT BE CHANGED)

Once a tuple is created, its items cannot be modified, added or removed.

```
t = (1, 2, 3)
t[0] = 10          # ❌ Error
t.append(4)        # ❌ Error
```

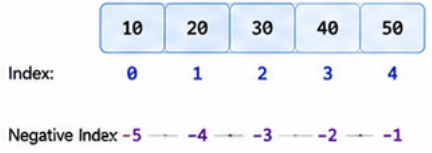
Note:
Tuples are immutable, but if a tuple contains mutable items, those items can be changed.

```
t = ([1, 2], 'a')
t[0].append(3)     # ✅ Allowed
print(t)           # ([1, 2, 3], 'a')
```

3. INDEXING

Access tuple items using index (starts from 0).

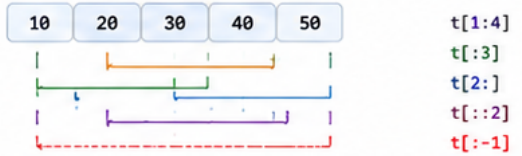
```
t = (10, 20, 30, 40, 50)
print(t[0])     # 10
print(t[2])     # 30
print(t[-1])    # 50
print(t[-2])    # 40
```



4. SLICING

Extract a part of the tuple.

```
t = (10, 20, 30, 40, 50)
print(t[1:4])      # (20, 30, 40)
print(t[:3])       # (10, 20, 30)
print(t[2:])       # (30, 40, 50)
print(t[::2])      # (10, 30, 50)
print(t[::-1])     # (50, 40, 30, 20, 10)
```



5. TUPLE METHODS

Method	Description	Example	Result
count()	Returns number of occurrences of a value	`t = (1, 2, 2, 3, 2)` `t.count(2)`	3
index()	Returns index of first occurrence of a value	`t = (10, 20, 30, 20)` `t.index(20)`	1

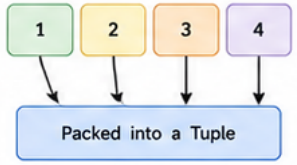
`index()` raises `ValueError` if the value is not found.

```
t.index(100)  # ❌ ValueError: 100 is not in tuple
```

6. PACKING

Packing means putting multiple values into a tuple.

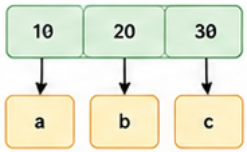
```
t = (1, 2, 3, 4)
print(t)     # (1, 2, 3, 4)
```



7. UNPACKING

Unpacking means extracting values from a tuple into variables.

```
t = (10, 20, 30)
a, b, c = t
print(a, b, c)    # 10 20 30
```



8. TUPLE INFO & PROPERTIES

```
t = (10, 'hello', 3.14, True)
print(type(t))    # <class 'tuple'>
print(len(t))     # 4
print(t)          # (10, 'hello', 3.14, True)
```

Property	Value
Data Type	tuple
Mutable?	No (Immutable)
Ordered?	Yes
Allows Duplicates?	Yes
Can be Dictionary Key?	Yes (if all items are immutable)

QUICK TIPS

- Use tuples when data should not be changed.
- Faster than lists.
- Useful for fixed data, coordinates, and dictionary keys.
- Use parentheses () to create tuples.

PRO TIP

If a tuple has only one item, add a comma.

```
t = (5,)      # Correct
type(t)       # <class 'tuple'>
t = (5)       # Not a tuple (int)
type(t)       # <class 'int'>
```

REAL WORLD USE CASES

- Storing fixed data (e.g., coordinates)
- Returning multiple values from a function
- Dictionary keys (if items are immutable)
- Database records (rows)
- Performance critical applications

NOTE

Tuples are immutable, but can contain mutable objects like lists.

```
t = ([1, 2], 3)
t[0].append(4)    # Allowed
```

SETS IN PYTHON

Unordered, Unique, and Mutable Collections

Unique Elements | Mutable (Changeable) | Unordered (No Index) | Fast Operations | Ideal for Membership Testing Testing

1. SET OVERVIEW

A set is an unordered collection of unique items.

- No duplicate values
- Unordered (no index/position)
- Mutable (you can add or remove items)
- Supports mathematical set operations

```
s = {1, 2, 3, 4}
print(type(s))     # <class 'set'>
print(s)           # {1, 2, 3, 4}
```

Sets do not allow duplicate values.

2. CREATING SETS

• Using curly braces {}

```
s = {1, 2, 3, 4}
print(s)          # {1, 2, 3, 4}
```

• Using set() constructor

```
s = set([1, 2, 3, 4, 4])
print(s)          # {1, 2, 3, 4}
```

• Empty set

```
empty_set = set()
print(type(empty_set))    # <class 'set'>
```

Note: {} creates an empty dictionary, use set() to create an empty set.

3. BASIC OPERATIONS

Operation	Example	Result
Membership	`2 in {1, 2, 3}` `5 in {1, 2, 3}`	True False
Length	`len({1, 2, 3})`	3
Iteration	`for x in {1, 2, 3}: print(x)`	1 2 3 (in any order)
Clear	`s = {1, 2, 3}` `s.clear()`	set()
Copy	`s2 = s.copy()`	{1, 2, 3}
Is Empty	`len(set()) == 0`	True

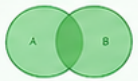
4. SET METHODS

Method	Description	Example	Result
add(x)	Adds an element x to the set	`s = {1, 2}` `s.add(3)`	{1, 2, 3}
remove(x)	Removes element x (raises error if not found)	`s = {1, 2, 3}` `s.remove(2)`	{1, 3}
discard(x)	Removes element x (if not found, no error)	`s = {1, 2, 3}` `s.discard(4)`	{1, 2, 3}
union(s)	Returns union of two sets (all unique items)	`{1, 2}.union({2, 3})` or `{1, 2} | {2, 3}`	{1, 2, 3}
intersection(s)	Returns common items in both sets	`{1, 2, 3}.intersection({2, 3, 4})` or `{1, 2, 3} & {2, 3, 4}`	{2, 3}
difference(s)	Items in first set but not in second set	`{1, 2, 3}.difference({2, 3, 4})` or `{1, 2, 3} - {2, 3, 4}`	{1}
symmetric_difference(s)	Items in either set but not in both	`{1, 2, 3}.symmetric_difference({2, 3, 4})` or `{1, 2, 3} ^ {2, 3, 4}`	{1, 4}

5. EXAMPLES OF SET OPERATIONS

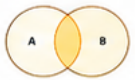
```
A = {1, 2, 3, 4}
B = {3, 4, 5, 6}
```

Union (A | B)
A.union(B)



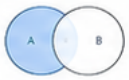
{1, 2, 3, 4, 5, 6}

Intersection (A & B)
A.intersection(B)



{3, 4}

Difference (A - B)
A.difference(B)



{1, 2}

Difference (B - A)
B.difference(A)

{5, 6}

Symmetric Difference (A ^ B)
A.symmetric_difference(B)

{1, 2, 5, 6}

6. SET COMPEHENSION

Create sets in a compact way.

```
squares = {x*x for x in range(1, 6)}
print(squares)          # {1, 4, 9, 16, 25}

even = {x for x in range(1, 11) if x % 2 == 0}
print(even)             # {2, 4, 6, 8, 10}
```

Benefits of Sets

- Fast membership testing (x in s)
- Remove duplicates easily
- Useful for mathematical operations
- Better performance for large data

7. REAL WORLD USE CASES

Remove Duplicate Data
Remove duplicate emails, user IDs, phone numbers etc.

Membership Testing
Check if a user exists in a system

Data Analysis
Find common/unique items in datasets

Tags & Categories
Store and manage unique tags

8. QUICK REFERENCE

Task	Code
Create set	`s = {1, 2, 3}`
Add item	`s.add(x)`
Remove item (error if not found)	`s.remove(x)`
Remove item (no error if not found)	`s.discard(x)`
Union	`s1.union(s2)` or `s1 | s2`
Intersection	`s1.intersection(s2)` or `s1 & s2`
Difference	`s1.difference(s2)` or `s1 - s2`
Symmetric Difference	`s1.symmetric_difference(s2)` or `s1 ^ s2`
Check membership	`x in s`
Set size	`len(s)`
Clear set	`s.clear()`

QUICK TIPS
- Sets remove duplicates automatically.
- Use sets for fast lookup and comparisons.
- Remember: No indexing or slicing in sets.

PRO TIP
Convert list to set to remove duplicates:
`unique_list = list(set(my_list))`

USE CASE EXAMPLE
```
common = users_A.intersection(users_B)
# Find common users between two groups
```

NOTE
Sets are unordered, so you cannot rely on the order of items.

DICTIONARIES IN PYTHON

Key-Value Pairs, Powerful & Flexible

Key-Value Pair | Mutable (Changeable) | Fast Lookup | Ordered (From Python 3.7+) | Diverse Data Types

1. DICTIONARY OVERVIEW

A dictionary (dict) stores data in key-value pairs.

- Keys must be unique and immutable (string, number, tuple)
- Values can be of any data type
- Mutable and dynamic in size

```
# Create a dictionary
d = {'name': 'Alice', 'age': 20, 'city': 'Delhi'}
print(d)
# Output: {'name': 'Alice', 'age': 20, 'city': 'Delhi'}
```

Keys act like an index. Each key maps to a specific value.

2. CREATING DICTIONARIES

- Using curly braces {}

```
d = {'name': 'Bob', 'age': 22}
```

- Using dict() constructor

```
d = dict(name='Bob', age=22)
```

- Empty dictionary

```
d = {}
print(type(d))  # <class 'dict'>
```

Note: Keys must be unique. If duplicate keys are used, later values overwrite earlier ones.

3. ACCESSING VALUES

Access values using keys.

```
d = {'name': 'Alice', 'age': 20, 'city': 'Delhi'}
print(d['name'])       # Alice
print(d['age'])        # 20
print(d.get('city'))   # Delhi
print(d.get('pin'))    # None (no error)
```

Method	Description
d[key]	Returns value for key (KeyError if key not found)
d.get(key, default)	Returns value if key exists, otherwise default

4. DICTIONARY METHODS

Method	Description	Example	Result
keys()	Returns all keys	d.keys()	dict_keys(['name', 'age', 'city'])
values()	Returns all values	d.values()	dict_values(['Alice', 20, 'Delhi'])
items()	Returns key-value pairs	d.items()	dict_items([('name', 'Alice'), ('age',20),('city', 'Delhi')])
get(key)	Returns value for key (without error)	d.get('age')	20
update(other)	Updates dictionary with another dict	d.update({'age': 21})	{'name': 'Alice', 'age': 21, 'city': 'Delhi'}
pop(key)	Removes key & returns its value	d.pop('age')	20
popitem()	Removes & returns last inserted key-value pair	d.popitem()	('city', 'Delhi')
clear()	Removes all items	d.clear()	{}

5. LOOPING THROUGH DICTIONARY

```
d = {'name': 'Alice', 'age': 20, 'city': 'Delhi'}

# Loop through keys
for k in d:
    print(k)          # name age city

# Loop through values
for v in d.values():
    print(v)          # Alice 20 Delhi

# Loop through key-value pairs
for k, v in d.items():
    print(f"{k} -> {v}")
```

items() is most useful for key-value access.

6. DICTIONARY OPERATIONS

- Check key exists

```
'age' in d        # True
```

- Add new item

```
d['pin'] = 110001
```

- Update value

```
d['age'] = 21
```

- Delete item

```
del d['city']
```

- Length of dict

```
len(d)
```

Example

```
d = {'name': 'Alice', 'age': 20}
'dob' in d            # False
d['dob'] = '01-01-2000'
d['age'] = 21
del d['name']
print(len(d))         # 2
print(d)
# {'age': 21, 'dob': '01-01-2000'}
```

7. NESTED DICTIONARY

Dictionary inside dictionary.

```
student = {
    'name': 'Rohit',
    'marks': {
        'math': 90,
        'science': 85
    }
}
print(student['name'])          # Rohit
print(student['marks']['math']) # 90
```

8. DICTIONARY COMPREHENSION (Python 3+)

Create dictionary using expression.

```
squares = {x: x*x for x in range(1, 6)}
print(squares)
# {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

9. DICTIONARY QUICK REFERENCE

Task	Code	Task	Code	Task	Code
Create dict	d = {} or dict()	Get all values	d.values()	Remove last item	d.popitem()
Access value	d[key]	Get all items	d.items()	Clear dictionary	d.clear()
Safe access	d.get(key, default)	Add/Update	d[key] = value	Get size	len(d)
Get all keys	d.keys()	Remove key	d.pop(key)		

QUICK TIPS

- Use get() to avoid KeyError.
- keys(), values(), items() return views.
- Dictionary is unordered (but maintains insertion order from Python 3.7+).
- Use clear() or {} to empty a dictionary.

PRO TIP

Use dictionaries when you need fast lookups by a key.

```
phonebook = {'Rahul': 1234567890}
print(phonebook['Rahul'])
# 1234567890
```

REAL WORLD USE CASES

- Storing user profiles
- Configuration settings
- Caching & lookups
- APIs & JSON data
- Counting occurrences

NOTE

Keys must be immutable and unique. Lists, sets or dictionaries cannot be used as keys.

DICTIONARY & LIST COMPREHENSION

Write Less Code, Do More!

Shorter Code | More Readable | Faster Development | Less Error Prone | Pythonic Way

1. WHAT IS COMPREHENSION?

Comprehension provides a concise way to create lists, dictionaries and sets.

- More readable
- Less code
- Often faster

Basic Syntax

```
List       : [x for x in items]
Dictionary : {key: value for item in items}
Set        : {x for x in items}
```

2. LIST COMPREHENSION

Basic Syntax

```
[expression for item in iterable]
```

Example

```
numbers = [1, 2, 3, 4, 5]
squares = {x*x for x in numbers]
print(squares)    # [1, 4, 9, 16, 25]
```

With Condition

```
even = [x for x in numbers if x % 2 == 0]
print(even)       # [2, 4]
```

3. DICTIONARY COMPREHENSION

Basic Syntax

```
{key_expr: value_expr for item in iterable}
```

Example

```
numbers = [1, 2, 3, 4]
squares_dict = {x: x*x for x in numbers}
print(squares_dict)
# {1: 1, 2: 4, 3: 9, 4: 16}
```

With Condition

```
even_dict = {x: x*x for x in numbers if x % 2 == 0}
print(even_dict)      # {2: 4, 4: 16}
```

4. SET COMPREHENSION

Basic Syntax

```
{expression for item in iterable}
```

Example

```
numbers = [1, 2, 2, 3, 3, 4]
unique_squares = {x*x for x in numbers}
print(unique_squares)    # {1, 4, 9, 16}
```

With Condition

```
odd_squares = {x*x for x in numbers if x % 2 != 0}
print(odd_squares)     # {1, 9}
```

5. COMPREHENSION WITH CONDITIONS

IF Condition

```
[x for x in range(10) if x % 2 == 0]
# [0, 2, 4, 6, 8]

{x: x*2 for x in range(5) if x > 2}
# {3: 6, 4: 8}

{x for x in range(10) if x % 3 == 0}
# {0, 3, 6, 9}
```

IF-ELSE Condition

```
["Even" if x % 2 == 0 else "Odd" for x in range(5)]
# ['Even', 'Odd', 'Even', 'Odd', 'Even']

{x: ('Even' if x % 2 == 0 else 'Odd')
  for x in range(5)}
# {0: 'Even', 1: 'Odd', 2: 'Even', 3: 'Odd', 4: 'Even'}
```

Multiple Conditions

```
[x for x in range(20) if x % 2
== 0 if x % 3 == 0}
# [0, 6, 12, 18]

{x for x in range(30) if x % 2
== 1 if x % 5 == 0}
# {5, 15, 25}
```

6. NESTED LOOPS IN COMPREHENSION

List Comprehension

```
matrix = [[1,2,3], [4,5,6]]
flat_list = {x for row in matrix
          for x in row]

print(flat_list)
# {1, 2, 3, 4, 5, 6]
```

Dictionary Comprehension

```
pairs = [(1,'a'), (2,'b'), (3,'c')]
pair_dict = {k: v for k, v in pairs}

print(pair_dict)
#  {1: 'a', 2: 'b', 3: 'c'}
```

Set Comprehension

```
matrix = [[1,2],[2,3]]
unique = {x for row in matrix
        for x in row}

print(unique)
# {1, 2, 3}
```

7. COMPONENTS OF COMPREHENSION

```
[ expression   for item in iterable   if condition ]
```

Expression	Loop	Condition (Optional)
What to return or compute	Iterate through items	Filter items (optional)

- Expression: The value or result you want in the new collection.
- Loop: Iterates over each item in iterable.
- Condition: Filters items (optional). If True, item is included.

8. QUICK COMPARISON

Type	Syntax Example	Result	Use Case
List	[x for x in items]	[1, 4, 9, 16]	Ordered collection
Dictionary	{x: x*x for x in items}	{1: 1, 2: 4}	Key-Value mapping
Set	{x*x for x in items}	{1, 4, 9}	Unique values

9. REAL WORLD EXAMPLES

- Get first letters of words: `[word[0] for word in ["apple", "banana", "cherry"]]`
- Create length dictionary: `{word: len(word) for word in ["cat", "dog", "elephant"]}`
- Unique characters from string: `{ch for ch in "programming"}`
- Square of even numbers (0-20): `[x*x for x in range(21) if x % 2 == 0]`

QUICK TIPS

- Use comprehensions for cleaner and shorter code.
- Add conditions to filter data easily.
- Great for data transformation and filtering.

PRO TIP

Comprehensions are not just shorter, they often run faster than loops! Always prefer Pythonic way.

BEST PRACTICES

- Keep expressions simple.
- Avoid complex logic inside comprehension.
- Use meaningful variable names.

USE CASES

- Data cleaning & transformation
- Filtering data
- Generating collections
- Building mappings

NOTE

Overusing comprehensions can reduce readability. Use them wisely!

FUNCTIONS IN PYTHON

Reusable Code Blocks • Organize • Reuse • Simplify

Reusability | Modularity | Efficiency | Readability | Maintainability

1. WHAT IS A FUNCTION?

A function is a block of code that performs a specific task. It can be used multiple times without rewriting the code.

- Avoid repetition
- Improve readability
- Easy to maintain

Syntax

```
def function_name(parameters):
    # code block
    return value    # optional
```

2. DEFINING A FUNCTION (def)

Use the def keyword to create a function.

```
def greet(name):
    print(f"Hello, {name}!")
```

Example

```
greet("CodeSardar")      # Hello, CodeSardar!
greet("Python")          # Hello, Python!
```

By default, a function returns None if no return statement is used.

3. PARAMETERS & ARGUMENTS

- Parameters are variables in function definition.
- Arguments are actual values passed to the function.

```
def add(a, b):
    return a + b

result = add(5, 10)     # 5 and 10 are arguments
print(result)           # 15
```

Term	Meaning
Parameters	Variables listed in function definition (a, b in example)
Arguments	Actual values passed while calling the function (5, 10 in example)

4. RETURN STATEMENT

The return statement sends a result back to the caller.

```
def square(x):
    return x * x

print(square(6))      # 36
print(square(9))      # 81
```

Note: A function can have multiple return statements, it will return as soon as it hits one.

5. DEFAULT ARGUMENTS

You can provide default values to parameters.

```
def greet(name, msg="Welcome"):
    return f"{msg}, {name}!"

print(greet("CodeSardar"))       # Welcome, CodeSardar!
print(greet("Python", "Hi"))     # Hi, Python!
```

Default arguments are used when no value is provided for that parameter.

6. *args (ARBITRARY ARGUMENTS)

*args allows you to pass a variable number of positional arguments as a tuple.

```
def total(*args):
    return sum(args)

print(total(1, 2, 3))            # 6
print(total(10, 20, 30, 40))     # 100
```

Collects extra positional arguments as a tuple

7. **kwargs (KEYWORD ARGUMENTS)

**kwargs allows you to pass a variable number of keyword arguments as a dictionary.

```
def show_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

show_info(name="CodeSardar", age=32, city="Amritsar")
# name: CodeSardar
# age: 32
# city: Amritsar
```

**kwargs	Collects extra keyword arguments as a dictionary

8. LAMBDA FUNCTIONS

Lambda creates small anonymous functions in a single line.

```
add = lambda a, b: a + b
multiply = lambda x, y: x * y

print(add(3, 5))          # 8
print(multiply(4, 6))     # 24
```

Best for short, simple operations.
Commonly used with map(), filter(), reduce().

9. FUNCTION SUMMARY TABLE

Feature	Syntax	Example
Define Function	def name(params):	def add(a, b): ...
Return Value	return value	return a + b
Default Arguments	def f(a=10)	def greet(name, msg="Hi")
Arbitrary Args	def f(*args)	f(1, 2, 3, 4)
Keyword Args	def f(**kwargs)	f(name="Jeet", age=25)
Lambda	lambda args: expr	lambda x: x * 2

10. PRACTICAL EXAMPLES

Example 1: Factorial

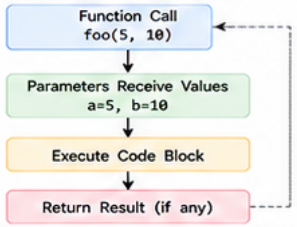
```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n-1)

print(factorial(5))    # 120
```

Recursive function calling itself.

Example 2: Calculator

```
def calc(a, b, op='+'):
    if op == '+': return a + b
    elif op == '-': return a - b
    elif op == '*': return a * b
    elif op == '/': return a / b

print(calc(10, 5))          # 15
print(calc(10, 5, '*'))     # 50
```

11. FLOW OF FUNCTION CALL



12. BEST PRACTICES

- Use meaningful function names
- Keep functions small and focused
- Use docstrings to explain purpose
- Use return instead of print
- Avoid global variables inside functions

Good functions make your code clean, readable and reusable!

QUICK TIPS

- Use functions to avoid repeating code.
- Break large problems into small functions.
- Test your functions with different inputs.

PRO TIP

Functions are first-class objects in Python.
You can assign, pass and return functions!
Example: `func = len`

REAL WORLD USE CASES

- Data processing & analysis
- Web development (APIs)
- Automation & scripting
- Machine Learning workflows

NOTE

Lambda functions are limited in complexity.
Use def for complex logic.

LAMBDA FUNCTIONS IN PYTHON

Small in Size, Big in Power!

Anonymous Function | Short Syntax | Fast & Lightweight | Commonly Used in Callbacks | Improves Code Readability

1. WHAT IS LAMBDA FUNCTION?

Lambda is a small anonymous function without a name.
It can have any number of parameters, but only one expression.

Syntax

```
lambda parameters: expression
```

- No function name
- Only one expression
- Returns the result automatically

2. BASIC SYNTAX

```
lambda arguments: expression
```

- `lambda` — Keyword
- `arguments` — Input values
- `expression` — Returned value

Lambda functions return the result of the expression automatically.

3. EXAMPLE: SIMPLE LAMBDA

```
# Add two numbers
add = lambda a, b: a + b
print(add(5, 3))      # 8
print(add(10, 20))    # 30

# Multiply two numbers
multiply = lambda x, y: x * y
print(multiply(4, 6))    # 24
```

4. LAMBDA COMPONENTS

Component	Description
lambda	Keyword to define lambda function
Parameters	Input values (can be zero or more)
Expression	Single expression that is evaluated
Return Value	Result of the expression is returned automatically

5. EXAMPLES WITH MULTIPLE PARAMETERS

```
# Subtract two numbers
sub = lambda a, b: a - b
print(sub(10, 4))   # 6

# Average of three numbers
avg = lambda x, y, z: (x + y + z) / 3
print(avg(6, 9, 15))      # 10.0

# Power (x raised to y)
power = lambda x, y: x ** y
print(power(2, 3))      # 8
```

6. LAMBDA WITH NO PARAMETERS

Lambda can also take no parameters.

```
greet = lambda: "Hello, CodeSardar!"
print(greet())   # Hello, CodeSardar!

get_pi = lambda:  3.14159
print(get_pi())     # 3.14159
```

7. LAMBDA WITH BUILT-IN FUNCTIONS (COMMON USE)

```
# With map() - double each number in list
nums = [1, 2, 3, 4]
doubled = list(map(lambda x: x * 2, nums))
print(doubled)         # [2, 4, 6, 8]

# With filter() - even numbers only
evens = list(filter(lambda x: x % 2 == 0, nums))
print(evens)         # [2, 4]

# With sorted() - sort by length of string
words = ['python', 'hi', 'code', 'lambda']
sorted_words = sorted(words, key=lambda x: len(x))
print(sorted_words)  # ['hi', 'code', 'python', 'lambda']
```

8. LAMBDA AS CALLBACK FUNCTION

Lambda functions are often used as callbacks.

```
from functools import reduce

# Find product of all numbers using reduce and lambda
nums = [1, 2, 3, 4]
product = reduce(lambda x, y: x * y, nums)
print(product)      # 24

# Using lambda with sort() for custom sorting
points = [(1, 2), (3, 1), (0, 5)]
points_sort(key=lambda p: p[1])
print(points)     # [(3, 1), (1, 2), (0, 5)]
```

9. LAMBDA vs NORMAL FUNCTION

Feature	Lambda Function	Normal Function
Name	No name (anonymous)	Has a name (`def`)
Syntax	Short and single line	Multi-line possible
Return	Implicit (expression)	Explicit (`return`)
Use	Small, simple tasks	Complex, large tasks
Readability	High for small tasks	High for big tasks

Use lambda for short, simple operations.
Use `def` for complex, multi-line logic.

10. PRACTICAL EXAMPLES

Task	Lambda Example	Output
Square of a number	`lambda x: x * x`	`9 (for x=3)`
Check even number	`lambda x: x % 2 == 0`	`True (for x=4)`
Concatenate strings	`lambda a, b: a + b`	`"Hello World"`
Find maximum	`lambda a, b: a if a > b else b`	`10 (for 10, 5)`
Reverse string	`lambda s: s[::-1]`	`"edoc" (for "code")`

10. QUICK REFERENCE

Use Case	Lambda Example
Add two numbers	`lambda a, b: a + b`
Square a number	`lambda x: x ** 2`
Check greater	`lambda a, b: a > b`
Length of string	`lambda s: len(s)`
Uppercase string	`lambda s: s.upper()`

COMMON BUILT-IN USES

- `map(func, iterable)` — Apply function to all items
- `filter(func, iterable)` — Filter items that return True
- `sorted(iterable, key=func)` — Sort using custom key
- `reduce(func, iterable)` — Reduce to a single value

SHORT SYNTAX EXAMPLES

```
inc = lambda x: x + 1                  # Increment
is_even = lambda x: x % 2 == 0         # Even check
area = lambda r: 3.14 * r * r          # Circle area
combine = lambda a, b, c: a + b + c    # Sum 3 numbers
first_char = lambda s: s[0]            # First character
```

QUICK TIPS

- Lambda is perfect for one-time small operations.
- Improves code readability in functional programming.
- Avoid using lambda for complex logic.
- Use meaningful parameter names.

PRO TIP

Lambda makes your code concise when used with `map()`, `filter()`, `sorted()` and `reduce()`.
Example: `list(map(lambda x: x*2, [1,2,3]))`

BEST PRACTICES

- Keep lambda short and simple.
- Use `def` for complex or multi-line functions.
- Use lambda in higher-order functions.

REAL WORLD USE CASES

- Data processing & transformation
- Filtering data
- Sorting with custom keys
- Event handling (callbacks)

NOTE

Lambda functions are limited to a single expression. For multiple steps, use `def` instead.

map()

Apply a function to all items of an iterable

Iterable	Function	Transform Data	List	Iterator	Result
Sequence of items	Takes item(s) & processes	Applies function to each item	Commonly used as output	map() returns an iterator	Transformed output

1. OVERVIEW

map() applies a given function to every item of an iterable (like list, tuple, etc.) and returns a map object (an iterator) with the results.

It does not return a list directly.
Convert it using `list()` or loop through it.

- Applies a function to all items
- Returns an iterator (map object)
- Lazy evaluation (processed on demand)
- Memory efficient

2. SYNTAX

```
map(function, iterable, ...)
```

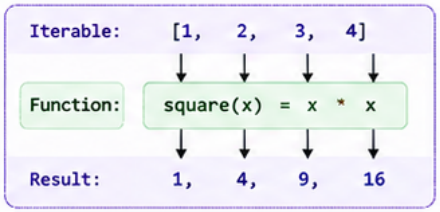
- **function** (applied to each item)
- **iterable** (sequence of items)

→ map object (iterator) (result of transformation)

You can pass multiple iterables.
The function is applied to items from all iterables in parallel.

3. HOW IT WORKS

1. Takes a function and an iterable.
2. Takes one item from the iterable.
3. Applies the function to that item.
4. Yields the result.
5. Repeats for all items.



4. EXAMPLES

```
# Convert list of strings to integers
nums = ['1', '2', '3', '4']
result = map(int, nums)
print(list(result))      # [1, 2, 3, 4]
```

```
# Square each number in a list
numbers = [1, 2, 3, 4]
squares = map(lambda x: x * x, numbers)
print(list(squares))     # [1, 4, 9, 16]
```

```
# Multiple iterables
a = [1, 2, 3]
b = [10, 20, 30]
added = map(lambda x, y: x + y, a, b)
print(list(added))       # [11, 22, 33]
```

5. PARAMETERS

Parameter	Description
function	The function to apply to each item. Can be a built-in function, user-defined function, or lambda.
iterable, ...	One or more iterables (list, tuple, etc.) map() works on items in parallel (like zip).

6. RETURN VALUE

map() returns a map object (an iterator).
It does not return a list.

```
result = map(function, iterable)
type(result)      # <class 'map'>
```

Use list(result), tuple(result), set(result) or loop to get the values.

7. EXAMPLE WALKTHROUGH

```
numbers = [1, 2, 3, 4]
result = map(lambda x: x*2, numbers)
```

Step	Item Taken	Function Applied	Result Yielded
1	1	1*2	2
2	2	2*2	4
3	3	3*2	6
4	4	4*2	8

Final Result → [2, 4, 6, 8]

8. QUICK REFERENCE

Task	Code
Use map()	`map(func, iterable)`
Convert to list	`list(map(func, iterable))`
Convert to tuple	`tuple(map(func, iterable))`
Use with lambda	`map(lambda x: x+1, data)`
Multiple iterables	`map(func, iter1, iter2, ...)`
Type of result	`<class 'map'> (iterator)`

QUICK TIPS

- map() is lazy → results are generated on demand.
- Use `list(map(...))` if you need the full list.
- Great for clean and concise code.
- Works with any iterable (list, tuple, string, etc.).

PRO TIP

Use map() with built-in functions for quick transformations.

```
list(map(str.upper, ['a','b','c']))
# ['A', 'B', 'C']
```

COMMON USE CASES

- Data type conversion (str → int, int → str)
- Mathematical operations
- String manipulations
- Applying same logic to multiple collections

NOTE

If multiple iterables are passed, map() stops when the shortest iterable is exhausted.

FILTER() FUNCTION

Keep Only What Matches the Condition

Condition Filtering | Iterable Input | True Values Only | Filter Data | Efficient & Short

1. WHAT IS filter()?

filter() is a built-in Python function that creates an iterator keeping only the items for which a condition is True.

`filter(function, iterable)`

- Applies a condition to each item
- Returns only True values
- Creates an iterator
- Memory efficient

2. SYNTAX

```
filter(function, iterable)
```

- **function** (condition)
- **iterable** (list, tuple, set, ...)

3. HOW IT WORKS

1. Take each item from the iterable.
2. Apply the condition (function).
3. If the condition returns True, keep the item.
4. Return an iterator of True values.

4. EXAMPLES

```
# Filter even numbers
nums = [1, 2, 3, 4, 5, 6]
even = filter(lambda x: x % 2 == 0 , nums)
print(list (even))      # [2, 4, 6]
```

```
# Filter names starting with 'A'
names = ['Alice', 'Bob', 'Ankit', 'Charlie']
a_names = filter(lambda n: n.starstwith('A'), names)
print(list(a_names))    # ['Alice', 'Ankit']
```

5. EXAMPLE WITH LAMBDA

```
numbers = [1, 2, 3, 4, 5, 6, 7]
result = filter(lambda x: x > 3, numbers)
print(list(result))       # [4, 5, 6, 7]
```

6. ITERABLE INPUT

filter() works with any iterable:

- List
- Tuple
- Set
- Dictionary (keys)
- String

7. TRUE VALUES

The condition can be any expression that returns True or False.

Examples:
- `x > 0`
- `len(x) > 3`
- `x.starstwith('a')`

8. RESULT

Returns an iterator, so you can use:

- `list(filter(...))` # Convert to list
- `for x in filter(...):` # Loop through
- `next(filter(...))` # Get first item

9. COMMON USE CASES

- Filter even/odd numbers
- Filter valid data
- Remove None values
- Get items by condition

10. MORE EXAMPLES

Filter None values

```
data = [1, None, 2, None, 3]
result = filter(None, data)
print(list(result))
# [1, 2, 3]
```

Filter strings by length

```
words = ['hi', 'python',
'js', 'code']
result = filter(lambda x: 2, words)
print(list(result))
# ['python', 'code']
```

Filter with def

```
def is_positive(x):
    return x > 0
nums = [-2, -1, 0, 1, 2]
result = filter(is_positive, nums)
print(list(result))      # [1, 2]
```

11. FILTER() vs LIST COMPREHENSION

Feature	filter()	List Comprehension
Syntax	filter(func, iter)	[x for x in iter if cond]
Returns	Iterator	List
Memory	Low (lazy)	High (eager)
Use case	Large data	Small data

12. QUICK TIPS

- Use filter() for clean and memory-efficient code.
- Convert result to list() if you need a list.
- Use lambda for simple conditions.

KEY POINTS:
- Takes function and iterable
- Keeps only True values
- Returns an iterator

PRO TIP:
Use filter(None, data) to remove None values from a list easily.

NOTE:
filter() returns an iterator, not a list. Use list() to get a list.

REAL WORLD USE:
- Filter user data, clean results,
- and process only valid information.

reduce() IN PYTHON

Reduce an iterable to a single value using a function

functools Module	Accumulator (Intermediate Result)	Iterable (Sequence of Items)	Combine Values Step by Step	Result (Single Value)	Function (Binary Function)

1. OVERVIEW

reduce() applies a function of two arguments cumulatively to the items of an iterable, from left to right, so as to reduce the iterable to a single value.

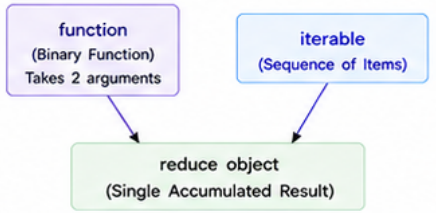
reduce() is not a built-in function. It is available in the functools module.

- Combines items using a function
- Returns a single accumulated result
- Works on any iterable (list, tuple, string, etc.)
- Applies function cumulatively (step by step)
- Memory efficient

2. SYNTAX

```
from functools import reduce
```

```
reduce(function, iterable[, initializer])
```

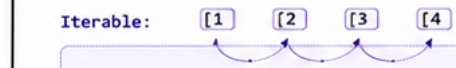


initializer is optional. It provides the initial value for the accumulator.

3. HOW IT WORKS

1. Take the first two items from the iterable.
2. Apply the function and store the result.
3. Use that result with the next item.
4. Repeat until all items are processed.
5. Return the final accumulated result.

Example Flow:




```
Step 1: f(1, 2) = 3
Step 2: f(3, 3) = 6
Step 3: f(6, 4) = 10

Result: 10
```

4. EXAMPLES

```
# Sum of numbers in a list
from functools import reduce
nums = [1, 2, 3, 4]
sum_result = reduce(lambda x, y: x + y, nums)
print(sum_result)          # 10
```

```
# Product of numbers in a list
nums = [1, 2, 3, 4]
prod_result = reduce(lambda x, y: x * y, nums)
print(prod_result)         # 24
```

```
# Concatenate strings
words = ['Hello', ' ', 'Python', ' ', 'World']
sentence = reduce(lambda x, y: x + y, words)
print(sentence)        # Hello Python World
```

5. PARAMETERS

Parameter	Description
function	A function that takes two arguments and returns a single value. It is applied cumulatively.
iterable	An iterable (list, tuple, string, etc.) whose items will be combined.
initializer (optional)	Initial value for the accumulator. If not provided, the first two items of the iterable are used.

6. RETURN VALUE

reduce() returns a single value – the final result after combining all items.

```
from functools import reduce
result = reduce(function, iterable[, initializer])
type(result)          # Depends on function
```

If the iterable is empty and initializer is not provided, reduce() raises TypeError.

7. EXAMPLE WALKTHROUGH

Find the sum using reduce()

```
nums = [1, 2, 3, 4]
from functools import reduce
result = reduce(lambda x, y: x + y, nums)
```

Step	Accumulator (x)	Next Item (y)	x + y	New Accumulator
Initial	1	2	1 + 2	3
1	3	3	3 + 3	6
2	6	4	6 + 4	10

Final Result → 10

8. QUICK REFERENCE

Task	Code
Import reduce	`from functools import reduce`
Sum of list	`reduce(lambda x, y: x + y, lst)`
Product of list	`reduce(lambda x, y: x * y, lst)`
With initializer	`reduce(lambda x, y: x + y, lst, 100)`
Find max	`reduce(lambda x, y: x if x > y else y, lst)`
Concatenate strings	`reduce(lambda x, y: x + y, lst)`
Type of result	`<class 'int'>, <class 'str'>, etc.`

QUICK TIPS

- Use reduce() for cumulative operations.
- Works great with lambda or custom functions.
- Always consider readability before using reduce().
- Use initializer for empty or edge cases.

PRO TIP

Use reduce() with built-in functions like operator.add, operator.mul for better performance.

```
from operator import add, mul
reduce(add, nums)      # Sum
reduce(mul, nums)      # Product
```

COMMON USE CASES

- Summing or multiplying values
- Finding maximum or minimum
- Concatenating strings
- Flattening nested structures
- Custom cumulative operations

EXAMPLE (REAL WORLD)

Calculate total price from cart items

```
cart = [100, 250, 150, 300]
total = reduce(lambda x, y: x + y, cart)
print(total)     # 800
```

NOTE

reduce() is powerful but can be hard to read for beginners. Use it when the logic is simple and cumulative.

enumerate() IN PYTHON

Loop over an iterable with an automatic counter (index)

Index	Value	Start	Loop	Iterable	Counter	Tuple
Auto Counter	Current Item	Default 0	Easy Iteration	Sequence	Incrementing	(Index, Value)

1. OVERVIEW

enumerate() adds a counter to an iterable and returns an enumerate object of (index, value) pairs as tuples.

Very useful when you need both index and value while looping.

- Provides automatic index for each item
- Returns an enumerate object (iterator)
- Highly memory efficient
- Custom start index (optional)
- Works with any iterable

2. SYNTAX

```
enumerate(iterable, start=0)
```

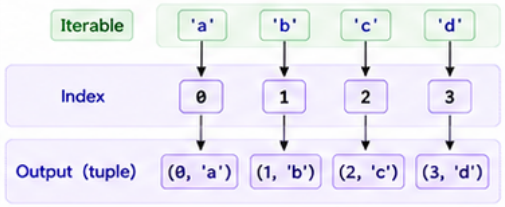
- **iterable** (Any iterable object: list, tuple, string, etc.)
- **start (optional)** (Starting index default is 0)

→ **enumerate object** (iterator of (index, value) tuples)

start = 0 by default.
Change it if you want indexing to begin from a custom number.

3. HOW IT WORKS

1. Takes an iterable and an optional start index.
2. Starts a counter at the start value.
3. For each item in the iterable:
   - Returns a tuple (index, value)
   - Increments the index by 1
4. Continues until all items are processed.
5. Returns an iterator of (index, value) tuples.



4. EXAMPLE

```
# Basic example
fruits = ['apple', 'banana', 'cherry']
for index, fruit in enumerate(fruits):
    print(index, fruit)
```

```
# Output:
# 0 apple
# 1 banana
# 2 cherry
```

```
# With custom start index
for i, fruit in enumerate(fruits, start=1):
    print(i, fruit)
```

```
# Output:
# 1 apple
# 2 banana
# 3 cherry
```

```
# With string
word = "Python"
for i, ch in enumerate(word):
    print(i, ch)
```

```
# Output:
# 0 P
# 1 y
# 2 t
# 3 h
# 4 o
# 5 n
```

5. PARAMETERS

Parameter	Type	Description	Default
iterable	iterable	Any iterable object (list, tuple, string, dict, set, etc.)	Required
start	int	Starting index for the counter	0

Only the first argument is required.
The start index helps when numbering should begin from a specific value.

6. RETURN VALUE

enumerate() returns an enumerate object (an iterator).

```
result = enumerate(iterable, start)
type(result)     # <class 'enumerate'>
```

- Each item is a tuple (index, value)
- Can be converted to list or used in loops

It does not store all items in memory.
It generates values on the go.

7. WALKTHROUGH

```
items = ['A', 'B', 'C']
for index, value in enumerate(items, start=5):
    print(index, value)
```

Step	Counter (index)	Value	Tuple Returned
1	5	'A'	(5, 'A')
2	6	'B'	(6, 'B')
3	7	'C'	(7, 'C')

```
Output:
5 A
6 B
7 C
```

8. QUICK REFERENCE

Task	Code
Basic usage	`enumerate(iterable)`
With custom start	`enumerate(iterable, start=1)`
Loop with index & value	`for i, x in enumerate(seq):`
Convert to list	`list(enumerate(seq))`
Get only index	`for i, _ in enumerate(seq):`
Get only value	`for _, x in enumerate(seq):`
Type of result	`<class 'enumerate'>` (iterator)
Each item type	`tuple` (index, value)

QUICK TIPS

- Use enumerate() when you need both index and value.
- Use _ if you don't need index or value.
- Save time and write cleaner loops.
- Works with any iterable, not just lists.

PRO TIP

enumerate() is more Pythonic than using range(len()).

```
# Avoid
for i in range(len(lst)):
    print(i, lst[i])
```

COMMON USE CASES

- Display numbered items in lists
- Process data with index
- Work with lines in files
- Track position in loops
- Pagination and numbering

EXAMPLE (REAL WORLD)

Print file lines with line numbers

```
with open('file.txt') as f:
    for line_no, line in enumerate(f, 1):
        print(f"{line_no}: {line.strip()}")
# Starts numbering from 1
```

NOTE

enumerate() returns an iterator. Once exhausted, it cannot be reused. Create a new one if needed.

zip() IN PYTHON

Combine elements from multiple iterables and return an iterator of tuples

- **Multiple Iterables** — Accepts two or more iterables
- **Pair Values** — Combines elements by position
- **Lists** — Works with lists
- **Tuples** — Works with tuples
- **Dictionary** — Works with dict keys
- **Loop** — Iterate over pairs easily
- **Iterator** — Returns a zip iterator

1. OVERVIEW

zip() takes multiple iterables and aggregates them in a tuple element-wise.
It returns an iterator of tuples.

It stops when the shortest iterable is exhausted. Use zip_longest() to continue till the longest.

- Combine multiple iterables
- Returns iterator of tuples
- Great for looping over related data
- Memory efficient
- Stops at the shortest iterable

2. SYNTAX

```
zip(*iterables)
```

- ***iterables** (One or more iterables)
- **Iterator** (Sequence of tuples)

→ zip object (iterator) (tuple of paired values)

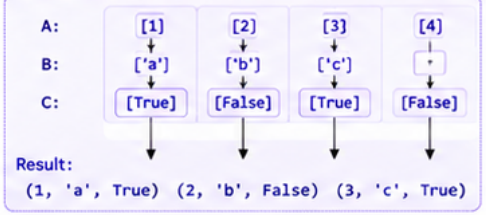
Use list() or loop to get the pairs.

```
list(zip(a, b, c))
```

3. HOW IT WORKS

1. Takes multiple iterables.
2. Takes the first item from each iterable.
3. Packs them into a tuple.
4. Moves to the next items.
5. Continues until the shortest iterable ends.

Example Flow:



4. EXAMPLES

```
# Zip two lists
nums = [1, 2, 3]
letters = ['a', 'b', 'c']
pairs = zip(nums, letters)
print(list(pairs))        # [(1, 'a'), (2, 'b'), (3, 'c')]
```

```
# Zip three iterables
a = [10, 20, 30]
b = ['x', 'y', 'z']
c = [True, False, True]
print(list(zip(a, b, c)))    # [(10, 'x', True),
                             (20, 'y', False), (30, 'z', True)]
```

```
# Zip with dictionary (keys)
d = {'name': 'Jeet', 'age': 32, 'city': 'Amritsar'}
print(list(zip(d.keys(), d.values())))
#  [('name', 'Jeet'), ('age', 32), ('city', 'Amritsar')]
```

5. PARAMETERS

Parameter	Description
*iterables	Two or more iterables (list, tuple, string, set, dict, etc.).

All iterables must be of the same length if you want all items paired.

6. RETURN VALUE

zip() returns a zip object (an iterator). Each item is a tuple containing elements from each iterable.

```
result = zip(a, b, c)
type(result)       # <class 'zip'>
```

Convert to list/tuple to see the output or use in a loop.

7. EXAMPLE WALKTHROUGH

```
names = ['Alice', 'Bob', 'Charlie']
ages  = [25, 30, 35]
scores = [88, 92, 79]
```

Step	names	ages	scores	Tuple Returned
1	'Alice'	25	88	('Alice', 25, 88)
2	'Bob'	30	92	('Bob', 30, 92)
3	'Charlie'	35	79	('Charlie', 35, 79)

Final Result:

```
[('Alice', 25, 88), ('Bob', 30, 92), ('Charlie', 35, 79)]
```

8. QUICK REFERENCE

Task	Code
Basic usage	`zip(a, b)`
Convert to list	`list(zip(a, b))`
Loop through	`for x, y in zip(a, b):`
Zip three lists	`zip(a, b, c)`
With dictionary	`zip(d.keys(), d.values())`
Handle different lengths	`itertools.zip_longest()`
Type of result	`<class 'zip'> (iterator)`

QUICK TIPS

- Use zip() for parallel iteration.
- Stop at shortest iterable.
- Use zip_longest() to include all items.
- Memory efficient, returns iterator.
- Unpack tuples directly in loops.

PRO TIP

Use asterisk * to unzip.

```
z = [(1, 'a'), (2, 'b'), (3, 'c')]
nums, letters = zip(*z)
print(list(nums))     # [1, 2, 3]
print(list(letters))  # ['a', 'b', 'c']
```

COMMON USE CASES

- Combine related data from lists
- Iterate over keys and values
- Transpose rows and columns
- Work with CSV / table data
- Unzipping data

EXAMPLE (REAL WORLD)

Create a table of student names and marks

```
students = ['Ram', 'Sam', 'John']
marks    = [85, 90, 78]
for name, mark in zip(students, marks):
    print(f"{name}: {mark}")
```

NOTE

zip() stops at the shortest iterable. Use zip_longest() from itertools to get all items with fillvalue.

any() & all() IN PYTHON

Return True if any or all elements in an iterable are True

Boolean Result | Conditions Check | Iterable Input | True Condition Met | False Condition Not Met | Validation Made Easy

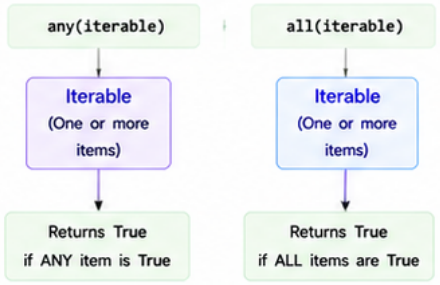
1. OVERVIEW

any() returns True if at least one element in the iterable is True.
all() returns True only if all elements in tho iterable are True.

Both functions work with any iterable (list, tuple, set, string, generator, etc.)

- any() → Checks if any element is True
- all() → Checks if all elements are True
- Returns a Boolean value (True / False)
- Short-circuits for efficiency
- Very useful for validation & conditions

2. SYNTAX



Empty iterable → any() returns False
Empty iterable → all() returns True (vacuous truth)

3. HOW IT WORKS

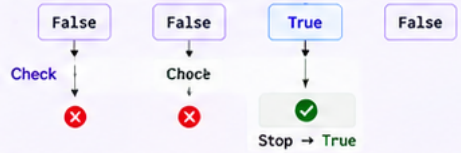
any()

1. Scans items one by one.
2. Returns True as soon as it finds the first True.
3. Returns False if no True is found.

all()

1. Scans items one by one.
2. Returns False as soon as it finds the first False.
3. Returns True if all items are True.

Example Flow (any: [False, False, True, False])



4. EXAMPLES

```
# any() examples
data1 = [False, False, True, False]
print(any(data1))           # True

data2 = [0, '', None, False]
print(any(data2))           # False

# all() examples
data3 = [1, 'yes', True, 10]
print(all(data3))           # True

data4 = [1, True, 0, 'ok']
print(all(data4))           # False

# With conditions
ages = [25, 18, 20, 17]
print(any(age < 18 for age in ages))      # True
print(all(age >= 18 for age in ages))     # False
```

5. PARAMETERS

Parameter	Type	Description
iterable	iterable	Any iterable object containing Boolean values or expressions that evaluate to Boolean.

Elements are evaluated in Boolean context.
Values like 0, '', None, False are False.
All other values are True.

6. RETURN VALUE

Both any() and all() return a Boolean value.

Function	Returns True When	Returns False When
any()	At least one item is True	All items are False (or iterable is empty)
all()	All items are True (or iterable is empty)	At least one item is False

Empty iterable → any() is False
Empty iterable → all() is True

7. EXAMPLE WALKTHROUGH

```
values = [0, False, '', 5, 'Hello']
print(any(values))    # True (5 is True)
print(all(values))    # False (0, False, '' are False)
```

Index	Value	Boolean Value	any() Check	all() Check
0	0	False	(continue)	False (Stop)
1	False	False	(continue)	False (Stop)
2	''	False	(continue)	False (Stop)
3	5	True	True (Stop)	-
4	'Hello'	True	-	-

Final Result → any: True, all: False

8. QUICK REFERENCE

Use Case	Use any()	Use all()
At least one condition must be True	✓	✗
All conditions must be True	✗	✓
Validate user input (any field filled)	✓	✗
Validate all inputs are correct	✗	✓
General existence check	✓	✗

QUICK TIPS

- any() is True if it finds first True.
- all() is False if it finds first False.
- Use generator expressions for large data.
- Great for validations and permissions.

PRO TIP

Use any()/all() with conditions:

```
numbers = [2, 4, 6, 8]
any(n > 5 for n in numbers)     # True
all(n % 2 == 0 for n in numbers)# True
```

COMMON USE CASES

- Form validation
- Permission checks
- Data quality checks
- Feature toggles
- Rule engine conditions

EXAMPLE (REAL WORLD)

```
# Check if any user is admin
users = [{'name':'A','admin':False},
         {'name':'B','admin':True}]
print(any(u['admin'] for u in users))  # True

# Check if all users are active
all_active = all(u['active'] for u in users)
```

NOTE

Remember: Empty iterable

any([]) → False

all([]) → True

python™ sorted() & reversed() IN PYTHON

Sort an iterable or reverse its order and return an iterator

Ascending	Descending	Key	Reverse	Iterable	Result
Low → High	High → Low	Custom Sort	Order	Input Data	Sorted Output

1. OVERVIEW

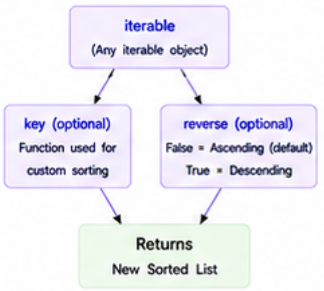
sorted() returns a new sorted list from the iterable object.
reversed() returns an iterator that iterates the iterable in reverse order.

Original iterable remains unchanged in both functions.

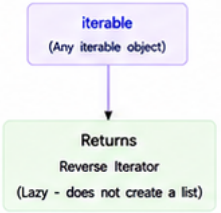
- sorted() → Returns a new sorted list
- reversed() → Returns a reverse iterator
- Supports any iterable (list, tuple, string, etc.)
- Use key to sort with custom logic
- Use reverse=True for descending order
- Memory efficient (reversed() is lazy)

2. SYNTAX

```
sorted(iterable, *, key=None, reverse=False)
```




```
reversed(iterable)
```



sorted() creates a new list (uses more memory).
reversed() returns an iterator (memory efficient).

3. HOW IT WORKS

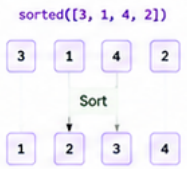
sorted()

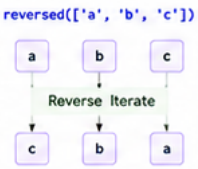
1. Takes an iterable.
2. Sorts the items.
3. Returns a new sorted list.

reversed()

1. Takes an iterable.
2. Iterates from last item to first item.
3. Returns an iterator.

Example Flow





4. EXAMPLES

```
# sorted() with numbers
nums = [4, 2, 9, 1, 7]
print(sorted(nums))                    # [1, 2, 4, 7, 9]
print(sorted(nums, reverse=True))      # [9, 7, 4, 2, 1]

# sorted() with strings
names = ['bob', 'Alice', 'charlie']
print(sorted(names))                   # ['Alice', 'bob', 'charlie']
print(sorted(names, key=str.lower))    # ['Alice', 'bob', 'charlie']

# sorted() with key (by length)
words = ['kiwi', 'apple', 'banana', 'fig']
print(sorted(words, key=len))          # ['fig', 'kiwi', 'apple', 'banana']

# reversed()
letters = ['a', 'b', 'c', 'd']
rev_iter = reversed(letters)
print(list(rev_iter))                  # ['d', 'c', 'b', 'a']

# reversed() with string
text = 'Python'
print(''.join(reversed(text)))         # 'nohtyP'
```

5. PARAMETERS (sorted())

Parameter	Type	Description	Default
iterable	iterable	Any iterable object (list, tuple, set, string, etc.)	Required
key	function	Function that takes one argument and returns a value used for sorting	None
reverse	bool	If True, sort in descending order	False

* indicates that key and reverse are keyword-only arguments.

6. RETURN VALUE

sorted() returns a new sorted list.
Original iterable is not modified.

```
result = sorted(iterable, key=Nono, reverse=False)
type(result)      # <class 'list'>
```

reversed() returns a reverse iterator.

```
result = reversed(iterable)
type(result)      # <class 'list_reverseiterator'>
```

Convert reversed iterator to list/tuple/string to see output.

7. EXAMPLE WALKTHROUGH

sorted() example

```
data = [5, 2, 9, 1, 7]
result = sorted(data, reverse=True)
```

Step	Action	List
1	Original List	[5, 2, 9, 1, 7]
2	Sort Descending	[9, 7, 5, 2, 1]
3	Return New List	[9, 7, 5, 2, 1]

Final Result: [9, 7, 5, 2, 1]

reversed() example

```
data = ['a', 'b', 'c', 'd']
rev_iter = reversed(data)
```

Step	Action	Output
1	Start from End	'd'
2	Move Back	'c'
3	Move Back	'b'
4	Move Back	'a'

Final Result: ['d', 'c', 'b', 'a']

8. QUICK REFERENCE

Task	sorted()	reversed()
Basic usage	sorted(iterable)	reversed(iterable)
Ascending order	sorted(iterable)	-
Descending order	sorted(iterable, reverse=True)	-
Custom sort (key)	sorted(iterable, key=func)	-
Get reverse order	-	reversed(iterable)
Convert to list	list(sorted(iterable))	list(reversed(iterable))
Return type	list	iterator
Original modified?	No	No

QUICK TIPS

- Use sorted() when you need a new list.
- Use reversed() for memory efficiency.
- Use key=len for sorting by length.
- Use reverse=True for descending order.

PRO TIP

You can chain both functions!

```
data = [3, 1, 4, 2]
print(list(reversed(sorted(data))))
    # [4, 3, 2, 1]
```

COMMON USE CASES

- Sorting numbers or strings
- Sorting objects by custom fields
- Getting reverse order of data
- Ranking, leaderboards, reports

EXAMPLE (REAL WORLD)

```
Sort students by marks (descending)
students = [
    {'name': 'Sam', 'marks': 88},
    {'name': 'Ram', 'marks': 95},
    {'name': 'John', 'marks': 72} ]
sorted_students = sorted(students, key=lambda x: x['marks', reverse=True)
for s in sorted_students:
    print(s['name'], s['marks'])
```

NOTE

sorted() is stable – items with equal key values keep their relative order.
reversed() works with any iterable (even generators).

MODULES IN PYTHON

Organize code • Reuse functionality • Write better Python programs

Module	import	from	import as	Built-in Module	Custom Module	Functions	Variables
Code Library	Load Module	Import Specific	Give Alias	Python Provided	User Defined	Reusable Code	Store Data

1. OVERVIEW

A module is a Python file containing definitions and statements.
It helps to organize code into reusable and maintainable parts.

Python has many built-in modules.
You can also create your own modules.

- Improves code reusability
- Makes code organized
- Easy to maintain
- Avoids repeating code
- Supports large projects

2. IMPORT SYNTAX

```
import module_name
```
Import entire module
```
import math
```

```
from module_name import name
```
Import specific name(s) from module
```
from math import sqrt
```

```
import module_name as alias
```
Import module with an alias (short name)
```
import numpy as np
```

Use modules to access ready-made functions and variables.

3. TYPES OF MODULES

Built-in Modules (Comes with Python)

math	random	datetime	os
Math functions	Random values	Date & Time	OS operations

sys	re	json	statistics
System info	Regular Expressions	JSON Handling	Statistical functions

Custom Modules (User Defined)

You can create your own modules (.py files) and import them in your programs.
Example: mymodule.py

4. HOW MODULES WORK

1 Write Module → 2 Import Module → 3 Use Members → 4 Get Result

.py file → Load in Program → Functions / Variables → Output

Example Flow

```
# Import module
import math

# Use function from module
result = math.sqrt(16)
print(result)          # 4.0
```

→ Output: `4.0`

5. MODULE MEMBERS

Modules can contain functions, variables, classes and objects.

Functions
Reusable blocks of code

```
def greet():
    print("Hello")
```

Variables
Store data inside module

```
PI = 3.14159
NAME = "Python"
```

Access using dot (.) operator
`module_name.member_name`

6. EXAMPLE: USING BUILT-IN MODULE

math Module

```
import math
print(math.pi)          # 3.14159
print(math.sqrt(25))    # 5.0
print(math.factorial(5)) # 120
```

random Module

```
import random
print(random.randint(1, 10))
print(random.choice(['A','B','C']))
print(random.random())  # 0.0--1.0
```

Explanation

- math → provides mathematical functions
- random → generates random values

7. EXAMPLE: CUSTOM MODULE

mymodule.py

```
PI = 3.14159

def circle_area(r):
    return PI * r * r

def greet(name):
    return f"Hello, {name}!"
```

main.py

```
import mymodule

print(mymodule.PI)
print(mymodule.circle_area(5))
print(mymodule.greet("Jeet"))
```

Output

```
3.14159
78.53975
Hello, Jeet!
```

8. QUICK REFERENCE

Statement	Meaning	Example
Import module	Import entire module	`import math`
from module import name	Import specific name	`from math import sqrt`
Import module as alias	Import with alias	`import numpy as np`
module.function()	Call function	`math.sqrt(16)`
module.variable	Access variable	`math.pi`

Use dir(module_name) to see all members in a module.

QUICK TIPS

- Import only what you need.
- Use alias for long module names.
- Check documentation with help(module_name).
- Keep your modules organized.

PRO TIP

Use aliases to write clean and short code.

```
import pandas as pd
df = pd.read_csv('data.csv')
```

COMMON USE CASES

- Math & calculations → math, statistics
- Data handling → json, csv, pandas
- File & OS operations → os, sys, shutil
- Web & API → requests
- Data analysis → numpy, pandas

EXAMPLE (REAL WORLD)

Read a JSON file and parse data

```
import json
with open('data.json') as f:
    data = json.load(f)
print(data['name'])      # Output name
```

NOTE

Custom module file name should be same as module name.
Place it in the same folder or Python path.

python™ PYTHON PACKAGES EXPLAINED

Organize • Reuse • Share Your Code

Package	__init__.py	Modules	Subpackages	import	pip	Library	Reusable Code
Collection of Modules	Marks Folder as Package	Python Files (.py)	Packages inside Packages	Use Code in Your Program	Install External Packages	Ready-made Solutions	Write Once Use Anywhere

1. WHAT IS A PACKAGE?

A package is a directory (folder) that contains one or more modules and a special file __init__.py.

It helps to organize related code into a structured and reusable way.

Packages allow you to split large projects into smaller, manageable parts.

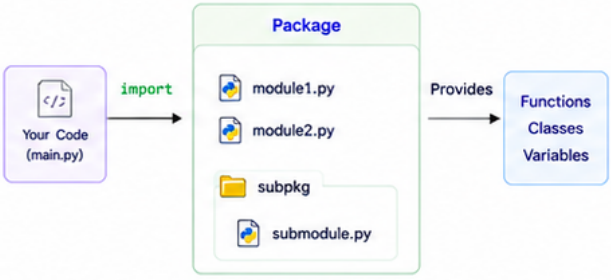
- Better Code Organization
- Reusable Across Projects
- Avoids Name Conflicts
- Easy to Maintain and Scale
- Share Your Code with Others

2. PACKAGE STRUCTURE

```
my_package/
├── __init__.py (marks this folder as a package)
├── module1.py
├── module2.py
└── subpkg/
    ├── __init__.py
    └── submodule.py
```

Every package folder must contain __init__.py (can be empty). It tells Python that this is a package.

3. HOW PACKAGES WORK



You import modules from a package and use the functions, classes and variables defined in them.

4. IMPORTING FROM PACKAGES

```python
# Import entire module
import my_package.module1

# Import specific function
from my_package.module1 import hello

# Import with alias
import my_package.module2 as m2

# Import from subpackage
from my_package.subpkg.submodule import greet
```

5. EXAMPLE: CREATE A PACKAGE

```
my_package/
├── __init__.py        → Marks as Package
├── math_utils.py      → Module
├── string_utils.py    → Module
└── subpkg/            → Subpackage
    ├── __init__.py    → Marks as Package
    └── converter.py   → Module in Subpackage
```

You can add as many modules and subpackages as needed.

6. EXAMPLE: USING PACKAGE

main.py

```python
from my_package import math_utils
from my_package.string_utils import shout
from my_package.subpkg.converter import km_to_m

print(math_utils.add(4, 5))        # 9
print(shout("hello"))              # HELLO!
print(km_to_m(2))                  # 2000
```

Output

```
9
HELLO!
2000
```

7. INSTALLING PACKAGES WITH pip

pip is Python package installer. It lets you install thousands of packages from Python Package Index (PyPI).

Common Commands

Command	Description
pip install package_name	Install a package
pip install package_name==1.2.3	Install specific version
pip uninstall package_name	Uninstall a package
pip list	List installed packages
pip show package_name	Show package details
pip search keyword	Search packages

Example: `pip install requests`

8. BUILT-IN vs CUSTOM PACKAGES

Built-in Packages

- Comes with Python
- No need to install
- Examples: math, os, sys, datetime, json

Custom / Third-Party Packages

- Install using pip
- Created by community or you
- Examples: requests, numpy, pandas, flask

Use built-in packages for common tasks. Install third-party packages for advanced features.

QUICK TIPS

- Always add __init__.py in your package folder.
- Use meaningful names for modules and packages.
- Keep your package structure clean.
- Write docs (README.md) to explain usage.

PRO TIP

You can make your own package and upload it to PyPI so others can install it!

```
python setup.py sdist bdist_wheel
# Then upload to PyPI
```

COMMON USE CASES

- Web Development (Flask, Django)
- Data Science (NumPy, Pandas)
- Machine Learning
- Automation & Scripting
- API Requests & Web Scraping

EXAMPLE (REAL WORLD)

Install and use requests package to make HTTP requests.

```python
import requests
res = requests.get('https://api.github.com')
print(res.status_code)    # 200
```

NOTE

Keep your virtual environment clean and install only the packages you need for your project.

pip IN PYTHON

Python Package Installer

WHAT IS pip?

pip is the package installer for Python.
It helps you install and manage packages from Python Package Index (PyPI).

WHY USE pip?

- Easy to install packages
- Manage dependencies
- Save time and effort
- Works with PyPI & custom sources

1. pip install

Install a package from PyPI.

Syntax

```
pip install <package_name>
```

Example

```
$ pip install requests
```

2. pip uninstall

Uninstall a package from your system.

Syntax

```
pip uninstall <package_name>
```

Example

```
$ pip uninstall requests
```

3. pip list

List all installed packages.

Syntax

```
pip list
```

Example

```
$ pip list
```

4. pip show

Show detailed information about a package.

Syntax

```
pip show <package_name>
```

Example

```
$ pip show requests
```

5. pip freeze

List installed packages in a requirements format.

Syntax

```
pip freeze
```

Example

```
$ pip freeze
```

6. pip upgrade

Upgrade a package to the latest version.

Syntax

```
pip install --upgrade <package_name>
```

Example

```
$ pip install --upgrade requests
```

7. requirements.txt

File that lists all required packages for a project.

Usage

```
pip install -r requirements.txt
```

Example

```
$ pip install -r requirements.txt
```

8. Packages

Reusable Python code libraries.

Common Sources

- PyPI (https://pypi.org)
- Custom Index / Private Repositories

EXAMPLE WORKFLOW

1. Install	2. Check Installed	3. Package Info	4. Upgrade	5. Freeze	6. Share File	7. Install All
pip install requests	pip list	pip show requests	pip install --upgrade requests	pip freeze > requirements.txt	requirements.txt	pip install -r requirements.txt

EXAMPLE: requirements.txt

```
requests==2.31.0
numpy==1.26.4
pandas==2.2.2
matplotlib==3.8.4
python-dotenv==1.0.1
```

Share this file with others so they can install the same packages easily.

COMMON OPTIONS

-U, --upgrade	Upgrade package
-r, --requirement	Install from requirements file
-y	Yes to all prompts
--user	Install for current user
--version	Show pip version
-h, --help	Show help

CHECK pip VERSION

```
pip --version
```

Example Output

```
pip 24.0
from C:\Python312\Lib\site-packages\pip
(python 3.12)
```

QUICK TIPS

- Always use virtual environments.
- Keep your packages updated.
- Use requirements.txt for teamwork.
- Check package details before installing.

PRO TIP

Create a virtual environment first:

```
python -m venv venv
venv\Scripts\activate      # Windows
source venv/bin/activate   # Mac/Linux
```

USEFUL RESOURCES

- PyPI: https://pypi.org
- pip Docs: https://pip.pypa.io/en/stable/
- Python Docs: https://docs.python.org/

COMMON USE CASES

- Web Development (Flask, Django)
- Data Science (NumPy, Pandas)
- Automation (Selenium, Requests)
- Machine Learning (Scikit-learn, TensorFlow)

NOTE

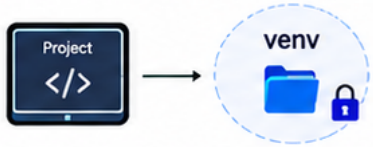
Running pip as root can cause permission issues. Use virtual environments whenever possible.

VIRTUAL ENVIRONMENT (venv)

Isolated Environment for Your Python Projects

- **Isolated Environment** — Keep projects separate
- **Dependencies** — Manage packages independently
- **Project** — Project-specific configuration
- **Environment** — Multiple environments per system

1. WHAT IS A VIRTUAL ENVIRONMENT?



A virtual environment is an isolated Python environment for a specific project. It has its own Python interpreter and installed packages, separate from the global system.

- Avoid dependency conflicts
- Keep projects clean and portable
- Share project without system dependencies
- Easy to reproduce environments

Best Practice: Always use virtual environments for every Python project.

2. CREATE VIRTUAL ENVIRONMENT

Using venv (built-in module)

```
# Navigate to your project folder
cd my_project

# Create virtual environment
python -m venv venv
```

This will create a folder named 'venv' in your project directory.

```
my_project/
├── venv/
│   ├── Scripts/      (Windows)
│   ├── bin/          (macOS/Linux)
│   ├── lib/
│   └── pyvenv.cfg
└── your_project_files.py
```

3. ACTIVATE VIRTUAL ENVIRONMENT

Windows (Command Prompt)

```
venv\Scripts\activate.bat
```

Windows (PowerShell)

```
venv\Scripts\Activate.ps1
```

macOS / Linux (bash/zsh)

```
source venv/bin/activate
```

After activation, your prompt will show the environment name:

```
(venv) C:\my_project>
```

4. DEACTIVATE VIRTUAL ENVIRONMENT

Deactivate when you are done working on the project.

```
deactivate
```

This will return you to your system's global environment.

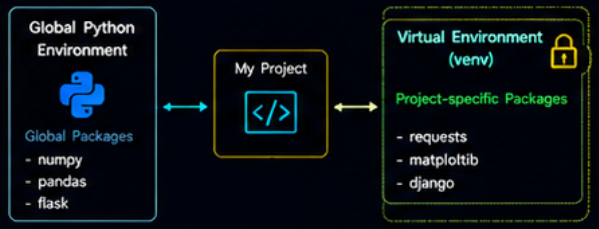
Check if DEACTIVATED

```
C:\my_project>
```

5. COMMON COMMANDS IN venv

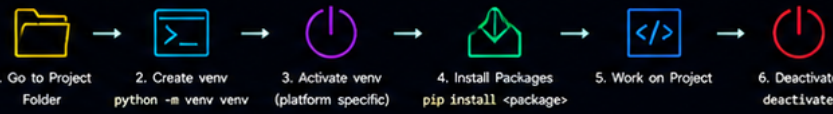
Command	Description
`python -m venv venv`	Create a new virtual environment
`source venv/bin/activate`	Activate (macOS/Linux)
`venv\Scripts\activate`	Activate (Windows CMD)
`deactivate`	Deactivate environment
`pip install package_name`	Install package in the environment
`pip list`	List installed packages
`pip freeze`	List packages in requirements format
`pip install -r requirements.txt`	Install packages from requirements.txt

6. HOW IT WORKS



Each project has its own environment and dependencies.
Changes in one environment do not affect others.

7. EXAMPLE WORKFLOW



requirements.txt Example

```
numpy==1.26.4
pandas==2.2.2
requests==2.32.3
matplotlib==3.8.4
```

Use `pip freeze > requirements.txt` to save your environment.
Share this file so others can install the same dependencies.

8. USEFUL TIPS

- Always activate the environment before installing packages.
- Add 'venv/' to .gitignore.
- Use requirements.txt for reproducible projects.
- Delete the venv folder to remove the environment.
- One project = One virtual environment.

BENEFITS

- No dependency conflicts
- Clean and isolated environments
- Easy project sharing
- Better project management

QUICK REFERENCE

Create : `python -m venv venv`
Activate : `source venv/bin/activate` (macOS/Linux)
`venv\Scripts/activate` (Windows)
Deactivate : `deactivate`

CHECK ACTIVE ENVIRONMENT

Check your prompt.
If you see `(venv)` at the beginning, your environment is active.

```
(venv) C:\my_project>
```

IMPORTANT NOTE

Never install packages globally for a project.
Always use a virtual environment!

REMEMBER

Virtual Environment =
Isolated • Organized • Reproducible
Better Python Development

FILE HANDLING IN PYTHON

Work with files easily and efficiently

- Store Data Permanently
- Read & Write File Content
- Safe & Secure Data Handling
- Easy Reuse of Data
- Organized Data Management

1. open()

Opens a file and returns a file object.

Syntax

```
open(file, mode, encoding)
```

Example

```
f = open("data.txt", "r",
    encoding="utf-8")
```

2. read()

Reads the entire content of the file.

Syntax

```
file.read(size)
```

Example

```
data = f.read()     # read all
data = f.read(10)   # read 10 chars
```

3. readline()

Reads a single line from the file.

Syntax

```
file.readlinee()
```

Example

```
line = f.readline()    # first line
line = f.readline()    # next line
```

4. readlines()

Reads all lines and returns a list.

Syntax

```
file.readlines()
```

Example

```
lines = f.readlines()
# list of lines
```

5. write()

Writes a string to the file.

Syntax

```
file.write(string)
```

Example

```
f.write("Hello Python!\n")
```

6. writelines()

Writes a list of lines to the file.

Syntax

```
file.writelines(list_of_lines)
```

Example

```
lines = ["Hello\n", "Python\n"]
f.writelines(lines)
```

7. close()

Closes the file. Free up resources.

Syntax

```
file.close()
```

Example

```
f.close()
```

8. with

Opens and closes the file automatically. (Recommended)

Syntax

```
with open(file, mode, encoding) as f:
    # operations
```

Example

```
with open("data.txt", "r", encoding="utf-8") as f:
    data = f.read()  # file closed automatically
```

FILE MODES

Mode	Description	If File Exists	If File Not Exists
'r'	Read	Open	Error
'w'	Write	Truncate	Create
'a'	Append	Append	Create
'x'	Create	Error	Create
'r+'	Read & Write	Open	Error
'w+'	Write & Read	Truncate	Create
'a+'	Append & Read	Open	Create
'x+'	Create & Read/Write	Error	Create

COMMON PARAMETERS

- `file` → File name with path (e.g., 'data.txt')
- `mode` → Mode of opening the file (see table)
- `encoding` → Character encoding (default: system encoding)

EXAMPLE: FILE HANDLING WORKFLOW

1. open() — Open the file
2. read() / readline() / readlines() — Read data
3. write() / writelines() — Write data
4. close() — Close the file

Always close the file after use to save changes and free resources.

EXAMPLE: COMPLETE EXAMPLE

```
# Writing to a file
with open("example.txt", "w", encoding="utf-8") as f:
    f.write("Hello Python!\n")
    f.writelines(["Line 2\n", "Line 3\n"])

# Reading from a file
with open("example.txt", "r", encoding="utf-8") as f:
    content = f.read()
    print(content)
```

Output

```
Hello Python!
Line 2
Line 3
```

The 'with' statement is the best way to handle files.

FILE POSITION (CURSOR)

The file cursor keeps track of the current position in the file.

Start — Current Cursor — End

Use methods:

- `file.tell()` → Get current position
- `file.seek(pos)` → Move cursor to position

COMMON USE CASES

- Reading configuration files
- Writing logs and reports
- Saving user data
- Processing large datasets
- Data analysis and automation

BEST PRACTICES

- Always use 'with' statement.
- Use proper file modes.
- Close files (automatically with 'with').
- Handle exceptions.

PRO TIP

Always specify encoding to avoid encoding errors.

```
open("file.txt", "r", encoding="utf-8")
```

EXCEPTION HANDLING

Use try-except while working with files.

```
try:
    f = open("data.txt", "r")
except FileNotFoundError:
    print("File not found!")
```

BINARY FILES

Use 'b' in mode for binary files.

```
f = open("image.png", "rb")
f = open("data.bin", "wb")
```

NOTE

Always test file operations and backup important data.

PYTHON FILE MODES

Different file modes to open files in Python

Read & Write File Content | Create / Modify Files | Add Data Safely | Prevent Data Loss | Binary Support for All Files

WHAT ARE FILE MODES?

File modes specify how a file will be opened and what operations can be performed on it.

TYPES OF MODES

Text Modes
- Work with text data (default mode)

Binary Modes
- Work with binary data (images, audio, etc.)

File Mode	Description	Operation	Creates File	Truncates File	Pointer Position	If File Exists	If File Not Exists	Example
r	Read (default)	Read only	✗	✗	Start of file	✓ Opens file	✗ Error	`open('file.txt', 'r')`
w	Write (overwrite)	Write only	✓ Yes	✓ Yes (clears content)	Start of file	✓ Overwrites content	✓ Creates new file	`open('file.txt', 'w')`
a	Append (add data)	Append (write at end)	✓ Yes	✗ No	End of file	✓ Appends data	✓ Creates new file	`open('file.txt', 'a')`
x	Create (exclusive)	Create new file only	✓ Yes	—	Start of file	✗ Error	✓ Creates new file	`open('file.txt', 'x')`
rb	Read Binary	Read only (binary)	✗	✗	Start of file	✓ Opens file	✗ Error	`open('file.bin', 'rb')`
wb	Write Binary (overwrite)	Write only (binary)	✓ Yes	✓ Yes (clears content)	Start of file	✓ Overwrites content	✓ Creates new file	`open('file.bin', 'wb')`
r+	Read + Write	Read & Write	✗	✗	Start of file	✓ Opens file	✗ Error	`open('file.txt', 'r+')`

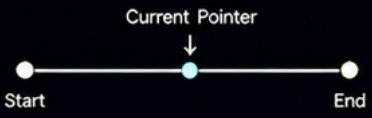
QUICK NOTES

- Use 'r' for reading file content.
- Use 'w' to write and overwrite existing content.
- Use 'a' to add new data without deleting old content.
- Use 'x' to create a new file only if it does not exist.
- Use 'b' with mode for binary files (images, audio, etc.).

EXAMPLES

```
open('file.txt', 'r')    | # Read file
open('file.txt', 'w')    | # Write (overwrite) file
open('file.txt', 'a')    | # Append to file
open('file.txt', 'x')    | # Create new file
open('image.png', 'rb')  | # Read binary file
open('data.bin', 'wb')   | # Write binary file
open('file.txt', 'r+')   | # Read and write file
```

POINTER POSITION EXPLAINED



The file pointer moves based on the operation and mode used.

BEST PRACTICES

- Always close the file after use.
- Use 'with' statement for safe handling.
- Choose the right mode for the right task.
- Back up important data before writing.

```
with open('file.txt', 'r') as f:
    data = f.read()
```

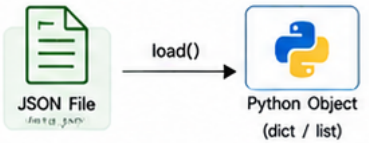
JSON IN PYTHON

Work with JSON data easily

JSON	JSON File	Python Object	API Data	Easy & Fast
Lightweight Data Format	Store & Share Data	Dict, List, Str, Int, etc.	Get & Send Data	Human Readable & Fast Parsing

1. json.load()

Read JSON data from a file



Example

```
import json
with open('data.json', 'r', encoding='utf-8') as f:
    data = json.load(f)
print(data)
```

Use When: Data is stored in a JSON file.

2. json.loads()

Parse JSON from a string

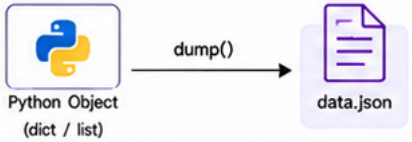
Example

```
import json
json_str = '{"name": "Python", "ver": 3}'
data = json.loads(json_str)
print(data)
```

Use When: JSON data is in string format.

3. json.dump()

Write Python object to a JSON file



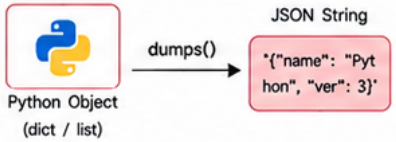
Example

```
import json
data = {"name": "Python", "ver": 3}
with open('data.json', "w", encoding='utf-8') as f:
    json.dump(data, f, indent=4)
print("Data written to file!")
```

Use When: Save Python object to a file.

4. json.dumps()

Convert Python object to JSON string



Example

```
import json
data = {"name": "Python", "ver": 3}
json_str = json.dumps(data, indent=4)
print(json_str)
```

Use When: Convert object to JSON string.

5. JSON File Example (data.json)

Sample JSON File Content

```
{
  "name": "Python",
  "ver": 3,
  "features": ["Easy", "Powerful", "Popular"],
  "active": true
}
```

Python Object (After load)

```
{
  'name': 'Python',
  'ver': 3,
  'features': ['Easy', 'Powerful', 'Popular'],
  'active': True
}
```

JSON vs Python Object

JSON	Python Object
Object	dict
Array	list
String	str
Number	int / float
true / false	True / False
null	None

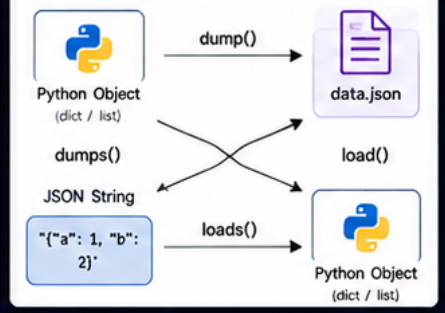
Common Use Cases

- **API Data** — Get data from REST APIs (JSON format)
- **Configuration Files** — Store app settings in JSON
- **Data Storage** — Save structured data in files
- **Data Exchange** — Share data between systems

Complete Example

```
# Write Python object to JSON file
import json
data = {"name": "Python", "ver": 3, "active": True}
with open('data.json', "w", encoding='utf-8') as f:
    json.dump(data, f, indent=4)

# Read JSON file
with open('data.json', 'r', encoding='utf-8') as f:
    data2 = json.load(f)
print(data2)

# Convert to JSON string
json_str = json.dumps(data2, indent=4)
print(json_str)
```

Workflow



Tips & Best Practices

- Always use `with open()` to handle files safely.
- Use `indent` for pretty JSON output.
- Use `ensure_ascii=False` for multi-language data.
- Validate JSON data before using.

Better Output with indent

```
json.dump(data, f, indent=4)
json.dumps(data, indent=4)
```

Pro Tip
JSON is language independent.
It is used by JavaScript, Python, Java, C#, etc.

Note
JSON supports only basic data types.
Custom objects must be converted.

Useful Module
`json` (built-in module)
No need to install anything!

Official Docs
https://docs.python.org/3/library/json.html

CSV IN PYTHON

Work with CSV files easily

- CSV Module Built-in
- CSV File Tabular Data
- Rows & Columns Structured Data
- Easy to Read & Write
- Lightweight & Fast

WHAT IS CSV?

CSV (Comma Separated Values) is a simple file format used to store tabular data in rows and columns.

CSV File (data.csv)

Name	Age	City
Rohit	25	Delhi
Sneha	22	Mumbai
Aman	28	Pune

Tabular Data (Rows & Columns)

Use When: Storing tabular data that can be opened in Excel/Sheets easily.

1. csv.reader()

Read CSV file row by row

CSV File (data.csv) → reader() → Rows (List of Lists)

```
import csv
with open('data.csv', 'r', newline='',
    encoding='utf-8') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)  # Each row is a list
```

Use When: You want to read CSV data row by row.

2. csv.writer()

Write rows to CSV file

Rows (List of Lists) → writer() → CSV File (output.csv)

```
import csv
data = [
    ['Name', 'Age', 'City'],
    ['Rohit', 25, 'Delhi'],
    ['Sneha', 22, 'Mumbai']
]
with open('output.csv', 'w', newline='',
          encoding='utf-8') as f:
    writer = csv.writer(f)
    writer.writerows(data)
```

Use When: You want to write data to CSV file.

3. csv.DictReader

Read CSV as dictionary

CSV File (data.csv) → DictReader → Dictionary (Each Row)

```
import csv
with open('data.csv', 'r', newline='',
    encoding='utf-8') as f:
    reader = csv.DictReader(f)
    for row in reader:
        print(row['Name'], row['Age'])
        # Each row is a dictionary
```

Use When: You want to access columns to header name.

4. csv.DictWriter

Write dictionary rows to CSV

Dictionary (Each Row) → DictWriter → CSV File (output.csv)

```
import csv
fieldnames = ['Name', 'Age', 'City']
data = [
    {'Name': 'Rohit', 'Age': 25, 'City': 'Delhi'},
    {'Name': 'Sneha', 'Age': 22, 'City': 'Mumbai'}
]
with open('output.csv', 'w', newline='',
    encoding='utf-8') as f:
    writer = csv.DictWriter(f, fieldnames=fieldnames)
    writer.writeheader()    # Write header
    writer.writerows(data)
```

Use When: You want to write dictionary data with headers.

5. CSV File Example

sample.csv

Name	Age	City
Rohit	25	Delhi
Sneha	22	Mumbai
Aman	28	Pune

Raw CSV Content

```
Name,Age,City
Rohit,25,Delhi
Sneha,22,Mumbai
Aman,28,Pune
```

CSV files can be opened in Excel, Google Sheets and many tools.

CSV TERMINOLOGY

Rows	Each line in CSV file (records)
Columns	Each field separated by comma (,)
Header	First row (optional) that contains column names
Data	Actual values in rows and columns

COMMON PARAMETERS

`newline=''`	Needed for proper reading/writing
`encoding='utf-8'`	Supports special characters
`delimiter=','`	Default comma (can be ; or others)
`quotechar='"'`	Character used for quoting strings

EXAMPLE WORKFLOW

CSV File (data.csv) → csv.reader() / DictReader → Process Data (Rows / Dicts) → csv.writer() / DictWriter → New CSV File (output.csv)

Read → Process → Write

Simple and Powerful!

SAMPLE CODE (Reader + Writer)

```
import csv

# Read CSV
with open('data.csv', 'r', newline='', encoding='utf-8') as f:
    reader = csv.DictReader(f)
    rows = list(reader)

# Modify Data
for row in rows:
    row['Age'] = int(row['Age']) + 1

# Write to New CSV
with open('output.csv', 'w', newline='', encoding='utf-8') as f:
    writer = csv.DictWriter(f, fieldnames=reader.fieldnames)
    writer.writeheader()
    writer.writerows(rows)
```

BEST PRACTICES

- Always use `newline=''` while opening files.
- Use DictReader/DictWriter for header based data.
- Use UTF-8 encoding for better compatibility.
- Validate data before writing to CSV.
- Handle exceptions for file operations.

PRO TIP

CSV is perfect for data exchange between systems and working with spreadsheets!

QUICK SUMMARY

- `csv.reader()` → Read rows as lists
- `csv.writer()` → Write rows from lists
- `DictReader` → Read rows as dictionaries
- `DictWriter` → Write dictionaries to CSV

USE CASES

- Exporting data to CSV for reports
- Importing data from CSV files
- Data migration between systems
- Working with Excel/Google Sheets data

COMMON FILE EXTENSION

.csv

Comma Separated Values

TOOLS THAT OPEN CSV

- Microsoft Excel
- Google Sheets
- Notepad / VS Code
- Pandas (Python)

NOTE

CSV does not support complex data types. Keep data clean and consistent for best results.

python™ EXCEPTION HANDLING IN PYTHON

Handle Errors Gracefully, Write Better Code

Exception	Error	Handling	Stability	Best Practice
Something went wrong	Program stops abnormally	Catch, Handle & Continue	Keeps program running	Write clean & reliable code

1. try

Place the code that might cause an exception inside try block.

try:

```
# Code that may cause error
result = 10 / 0
```

Example

```
try:
    x = int("abc")
    print(x)
except:
    pass
```

Use try to test a block of code for errors.

2. except

Catch and handle the exception if it occurs.

except:

```
# Handle the exception
print("An error occurred")
```

Example

```
try:
    x = int("abc")
except ValueError:
    print("Invalid value!")
```

Use multiple except blocks for different exceptions.

3. else

Runs if no exception occurs in try block.

else:

```
# Executes when no error
print("All good!")
```

Example

```
try:
    x = 10 / 2
except ZeroDivisionError:
    print("Can't divide by zero")
else:
    print("Result:", x)
```

Use **else** for code that should run only when no exception happens.

4. finally

Runs no matter what, exception happens or not.

finally:

```
# Always executes
print("Cleanup done")
```

Example

```
try:
    f = open("file.txt", "r")
    data = f.read()
except FileNotFoundError:
    print("File not found")
finally:
    print("Closing the file")
```

Use **finally** to release resources (close files, DB connections, etc).

5. raise

Manually raise an exception when needed.

raise

```
# Raise an exception
raise ValueError("Invalid!")
```

Example

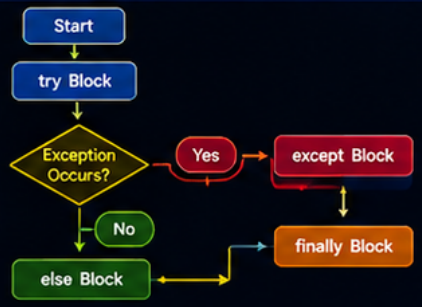
```
age = -5
if age < 0:
    raise ValueError("Age
        cannot be negative")
```

Use **raise** to trigger custom exceptions.

COMMON EXCEPTIONS

Exception	When it Occurs
ZeroDivisionError	Division by zero
ValueError	Invalid value/type
TypeError	Wrong data type
FileNotFoundError	File not found
KeyError	Key not found in dict
IndexError	List index out of range
ImportError	Module import fails
EOFError	End of file reached

EXCEPTION HANDLING FLOW



COMPLETE EXAMPLE

```
# Example: Read file safely
try:
    f = open("data.txt", "r")
    content = f.read()
except FileNotFoundError:
    print("File not found. Please check the name.")
except PermissionError:
    print("You don't have permission to read this file.")
else:
    print("File read successfully!")
    print(content)
finally:
    print("File operation completed.")
```

Output (If file not found)

```
File not found. Please check the name.
File operation completed.
```

CUSTOM EXCEPTION EXAMPLE

```
class NegativeAgeError(Exception):
    pass

age = -3
if age < 0:
    raise NegativeAgeError("Age cannot
    be negative!")
else:
    print("Age is valid")

# Handling custom exception
try:
    if age < 0:
        raise NegativeAgeError("Age cannot
        be negative!")
except NegativeAgeError as e:
    print("Custom Exception:", e)
```

BEST PRACTICES

- Catch specific exceptions, not generic ones.
- Keep try block small and focused.
- Always provide meaningful error messages.
- Use finally to clean up resources.
- Avoid using bare except:

```
# Bad
try:
    x = 10 / 0
except:
    pass    # Avoid this
```

QUICK SUMMARY

- try → Test risky code
- except → Handle the error
- else → No error? Run this
- finally → Always run this
- raise → Throw custom error

PRO TIP

Clear error handling = Happy users & stable code!

REMEMBER

Exceptions are not problems to fear, but events to handle!

COMMON USE CASES

- File operations
- User input validation
- API requests
- Database connections
- Data processing

NOTE

Unhandled exceptions crash the program. Handle them, don't ignore them!

KEY TAKEAWAY

Good exception handling makes your Python code robust, professional and production-ready.

python™ COMMON EXCEPTIONS IN PYTHON

Understand common errors and handle them like a pro!

Errors Happen Be Prepared | Handle Gracefully Avoid Crashes | Write Better Code Debug Faster | Know Exceptions Save Time

1 ValueError

Raised when a function gets an argument of the right type but an inappropriate value.

Example

```
age = int("abc")
# ValueError: invalid literal for int()
print(age)
```

When it occurs:
When a value is correct type but not a valid value.

2 TypeError

Raised when an operation or function is applied to an object of inappropriate type.

Example

```
result = "5" + 10
# TypeError: can only concatenate
# str (not "int") to str
```

When it occurs:
When the data type is not appropriate for the operation.

3 IndexError

Raised when trying to access an index that is out of range of a sequence.

Example

```
nums = [10, 20, 30]
print(nums[5])
# IndexError: list index out of range
```

When it occurs:
When an index is not valid for the sequence (list, tuple, string).

4 KeyError

Raised when a dictionary key is not found in the dictionary.

Example

```
data = {'name': 'Ram'}
print(data['age'])
# KeyError: 'age'
```

When it occurs:
When the specified key does not exist in a dictionary.

5 NameError

Raised when a local or global name is not found.

Example

```
print(x)
# NameError: name 'x' is not defined
```

When it occurs:
When a variable or function name is used before it is defined.

6 FileNotFoundError

Raised when a file or directory is requested but doesn't exist.

Example

```
f = open("data.txt", "r")
# FileNotFoundError: [Errno 2]
# No such file or directory: 'data.txt'
```

When it occurs:
When trying to open a file that is not found at the given path.

7 ZeroDivisionError

Raised when the second argument of a division or modulo operation is zero.

Example

```
a = 10 / 0
# ZeroDivisionError: division by zero
```

When it occurs:
When trying to divide or take modulo by zero.

8 ImportError

Raised when the imported module is not found.

Example

```
import not_a_module
# ImportError: No module named
# 'not_a_module'
```

When it occurs:
When the module you want to import does not exist.

Best Practices

- Always use try-except to handle exceptions.
- Don't ignore exceptions, handle them properly.
- Show meaningful error messages to users.
- Keep your code clean and error-free.

Exception Handling Example

```
try:
    num = int(input("Enter a number: "))
    print(10 / num)
except ZeroDivisionError:
    print("Cannot divide by zero!")
except ValueError:
    print("Please enter a valid number!")
else:
    print("Success!")
finally:
    print("Execution completed.")
```

Remember

Exceptions are not problems, they are events.

Handle them smartly, make your code robust!

Quick Tips

- Read the full error message carefully.
- Traceback helps you find the exact issue.
- Test your code with different inputs.
- Good error handling = Happy users 😊

Key Takeaway

Know exceptions, handle exceptions, write better Python code!

CUSTOM EXCEPTIONS IN PYTHON

Create your own exceptions for better error handling!

- Custom Error — Your Own Rules
- Better Handling — Specific Errors
- Clean Code — Easy to Maintain
- Clear Message — Easy to Debug
- Validation — Strong & Reliable

WHAT ARE CUSTOM EXCEPTIONS?

Custom exceptions allow you to create your own error types by extending the built-in Exception class. This helps in raising meaningful and application-specific errors.

WHY USE CUSTOM EXCEPTIONS?

- Handle specific application errors
- Provide clear and meaningful messages
- Improve code readability and maintenance
- Make debugging easier
- Follow clean and professional coding practices

HOW CUSTOM EXCEPTIONS WORK?

1. **Define Class** — Create a class that inherits from Exception
2. **Raise Error** — Use 'raise' to trigger your custom error
3. **Handle Error** — Use try-except block to handle your error
4. **Show Message** — Display a clear and helpful error message

EXAMPLE: SIMPLE CUSTOM EXCEPTION

```python
# Define custom exception
class MyError(Exception):
    def __init__(self, message):
        self.message = message
        super().__init__(self.message)

# Use the custom exception
def check_age(age):
    if age < 0:
        raise MyError("Age cannot be negative!")
# Test the code
try:
    check_age(-5)
except MyError as e:
    print(f"Custom Error: {e}")
```

OUTPUT

```
Custom Error: Age cannot be negative!
```

CUSTOM EXCEPTION ANATOMY

```python
class MyError(Exception):
    def __init__(self, message):
        self.message = message
        super().__init__(self.message)
```

- `class MyError(Exception):` — Class Name (Your Custom Error); Base Class (Inherits Exception)
- `def __init__(self, message):` — Constructor (Initializes Message)
- `super().__init__(self.message)` — Call Parent Constructor (Important for Python)

Note: Always call `super().__init__(message)` to ensure proper behavior of the exception.

EXAMPLE: VALIDATION WITH CUSTOM EXCEPTION

```python
# Custom exception for invalid email
class InvalidEmailError(Exception):
    pass

def validate_email(email):
    if "@" not in email or "." not in email:
        raise InvalidEmailError("Invalid email format!")
    return True

try:
    email = input("Enter email: ")
    if validate_email(email):
        print("Email is valid!")
except InvalidEmailError as e:
    print(f"Error: {e}")
except Exception as e:
    print(f"Unexpected error: {e}")
```

SAMPLE RUN

```
Enter email: johndoe.com
Error: Invalid email format!
----------
Enter email: john@
Email is valid!
```

Pro Tip — Use specific custom exceptions to give users clear feedback.

BUILT-IN vs CUSTOM EXCEPTIONS

Built-in Exception	When to Use	Custom Exception	Benefit
ValueError	Invalid value	InvalidAgeError	More specific
TypeError	Wrong type	InvalidTypeError	Clear purpose
IndexError	Index out of range	InvalidIndexError	Better readability
KeyError	Key not found	MissingKeyError	Domain specific
Exception	General errors	MyAppError	App-level control

BEST PRACTICES

- Use meaningful and specific error names.
- Inherit from Exception class.
- Provide clear error messages.
- Handle exceptions at the right level.
- Don't use bare except: (avoid this).

DO'S

- Use custom exceptions for business logic errors.
- Keep error messages user-friendly.
- Log the error for debugging.

DON'TS

- Don't overuse custom exceptions.
- Don't hide the real error.
- Don't ignore exceptions.

COMMON USE CASES

- Input validation (forms, data)
- Business rule violations
- Authentication & Authorization
- File & Resource handling
- API & Service layer errors

SUMMARY

Custom exceptions help you take control of error handling in your application. They make your code cleaner, more reliable, and easier to debug.

python DEBUGGING IN PYTHON

Find Errors, Understand Code, Fix Faster!

Find Errors Quickly | Understand Flow Better | Track Issues Easily | Fix Bugs Confidently | Write Better Code

1. Using print() for Debugging

The simplest way to debug.

```
x = [1, 2, 3]
print("Start")
for i in x:
    print("Value:", i)
print("End")
```

Use print() to check variable values and program flow.

2. Using pdb (Python Debugger)

Built-in debugger: step by step execution.

```
import pdb
def divide(a, b):
    pdb.set_trace()  # Start debugger
    return a / b

divide(10, 0)
```

PDB Prompt (Example)

```
> n     # next line
> s     # step into
> c     # continue
> p a   # print variable a
> q     # quit debugger
```

3. breakpoint() (Python 3.7+)

Modern way to start debugging.

```
def add(a, b):
    breakpoint()  # Pause here
    return a + b

add(5, 7)
```

Works same as pdb.set_trace() but cleaner and easier! Python 3.7+

4. Traceback (Error Details)

Shows where and why error happened.

```
Traceback (most recent call last):
  File "app.py", line 5, in <module>
    result = 10 / 0
ZeroDivisionError: division by zero
```

- Traceback shows file name, line number and error type.
- Helps you find the exact problem.

5. Common Errors You'll See

NameError	Variable or function name not found.
TypeError	Operation on wrong data type.
IndexError	Index out of range in list or string.
KeyError	Key not found in dictionary.
ZeroDivisionError	Division by zero not allowed.
FileNotFoundError	File or path not found.

6. Using Debugger in VS Code

Powerful debugging right in your editor.

1. Click on Run and Debug (Ctrl+Shift+D)
2. Click "create a launch.json file"
3. Add breakpoints (click on the left of line number)
4. Press F5 to start debugging
5. Use Debug Panel to control execution

Debug Controls

Continue (F5)	Step Over (F10)	Step Into (F11)	Step Out (Shift+F11)	Restart (Ctrl+Shift+F5)	Stop (Shift+F5)

7. Breakpoints (Pause Points)

Pause code at important lines.

script.py

```
1  def calculate(x, y):
2      result = x + y
3      print("Result:", result)
4      return result
5
6  calculate(5, 10)
```

Types of Breakpoints

- Red Dot Breakpoint — Pause on a line
- Conditional Breakpoint — Pause when condition is true
- Logpoint — Log message without stopping execution

Breakpoints help you inspect values, flow and find the bug easily.

8. Error Tracking Tips

- Read full error message carefully.
- Check the last line (error type).
- Look at line number and file name.
- Reproduce the error again.
- Use debugger to inspect values.
- Fix the issue and test again.

Pro Tip

Good debugging = 50% understanding + 50% patience!

9. Debugging Workflow

Write Code → Run Code → Error / Unexpected Output → Debug (Use tools) → Fix and Test → Repeat until perfect!

Best Practices

- Keep code small and test frequently.
- Use meaningful variable names.
- Add logs or print() where needed.
- Remove debug code before production.

Helpful Tools

- pdb (built-in)
- breakpoint()
- VS Code Debugger
- Logging Module

Keyboard Shortcuts (VS Code)

F5	Start / Continue
F10	Step Over
F11	Step Into
Shift + F11	Step Out
Ctrl + Shift + F5	Restart
Shift + F5	Stop

Remember

Bugs are not enemies, they are teachers.

Debug today, code better tomorrow! 💪

Summary

- Use print(), pdb, breakpoint() to debug.
- Read Traceback to find actual error.
- Breakpoints help to pause and inspect.
- Practice = Perfect Debugger!

python™ OBJECT-ORIENTED PROGRAMMING (OOP) IN PYTHON

Build scalable, reusable and organized code with OOP!

Reusability	Maintainability	Modularity	Scalability
Use code again & again	Easy to update and manage	Break code into smaller parts	Easy to grow and extend

WHAT IS OOP?

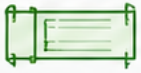
OOP (Object-Oriented Programming) is a programming paradigm based on the concept of Objects which contain data (attributes) and code (methods).

1. Class

A class is a blueprint for creating objects.

```
class Car:
    pass
```

Class: Car



2. Object

An object is an instance of a class.

```
car1 = Car()
car2 = Car()
```

Objects: car1, car2

3. Methods

Methods are functions defined inside a class.

```
class Car:
    def start(self):
        print("Car Started")
```

Call: car1.start()

Car Started

4. Properties

Properties (attributes) store data about objects.

```
class Car:
    def __init__(self, name):
        self.name = name
```

car = Car("BMW")

car.name → "BMW"

5. Constructor

Constructor (__init__) is called when object is created.

```
class Car:
    def __init__(self, name):
        self.name = name
        print("Object Created")
```

Called automatically when object is created.

Object Created

6. Inheritance

Inheritance allows a class to inherit attributes and methods from another class.

```
class Vehicle:
    def move(self):
        print("Moving...")

class Car(Vehicle):
    pass
```

Vehicle (Parent Class) → Car (Child Class)

7. Polymorphism

Polymorphism allows methods to do different things based on the object.

```
class Animal:
    def sound(self):
        pass

class Dog(Animal):
    def sound(self):
        print("Bark")

class Cat(Animal):
    def sound(self):
        print("Meow")
```

```
a1 = Dog()
a2 = Cat()

for a in (a1, a2):
    a.sound()
```

Output:
Bark
Meow

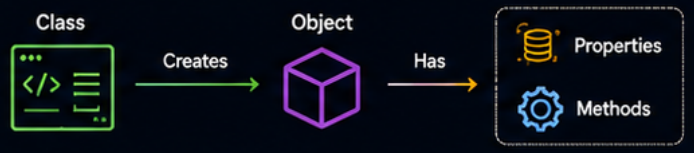
8. Encapsulation

Encapsulation hides data and allows access through methods.

```
class Account:
    def __init__(self, balance):
        self.__balance = balance  # private

    def deposit(self, amount):
        self.__balance += amount

    def get_balance(self):
        return self.__balance
```

__balance is private and can only be accessed using methods.

OOP IN PYTHON – AT A GLANCE



KEY BENEFITS

- Code Reusability
- Better Organization
- Easy Maintenance
- Flexible & Scalable
- Real-world Modeling

QUICK EXAMPLE

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def introduce(self):
        print(f"Hi, I am {self.name}, {self.age} years old.")
```

USAGE

```
p1 = Person("Rohit", 25)
p2 = Person("Sneha", 22)

p1.introduce()   # Hi, I am Rohit, 25 years old.
p2.introduce()   # Hi, I am Sneha, 22 years old.
```

IMPORTANT POINTS

- Class is a blueprint.
- Object is a real-world entity.
- Methods define the behavior.
- Attributes store the data.
- OOP makes code clean, modular and powerful.

REAL-WORLD EXAMPLE

- Car → Class
- BMW, Audi → Objects
- start(), stop() → Methods
- color, model → Properties

PRO TIP

Think in terms of real-world objects and design your code around them. This makes your programs more natural and easy to manage!

CLASSES & OBJECTS IN PYTHON

Blueprints for creating real-world things!

Organized Code | Reusability | Modularity | Easy to Maintaiin | Scalability

1. class

A class is a blueprint or template for creating objects.

```
class Person:
    pass
```

2. Object

An object is an instance of a class. It has its own data and behavior.

```
p1 = Person()
p2 = Person()
```

3. __init__()

The __init__() method is a constructor. It runs automatically when the object is created.

```
def __init__(self, name, age):
    self.name = name
    self.age = age
```

4. self

self refers to the current object instance. It is used to access attributes and methods.

```
self.name = name
self.age = age
```

5. Attributes

Attributes are variables that belong to an object (or class). They store data about the object.

```
self.name = name
self.age = age
```

6. Methods

Methods are functions defined inside a class. They define the behavior of objects.

```
def greet(self):
    print("Hello!")
```

7. Instance

Every object is an instance of a class. Each instance has its own data.

```
p1 and p2 are
instances of
Person class.
```

8. Create Object

Use class name followed by () to create an object.

```
person = Person()
# Object created
```

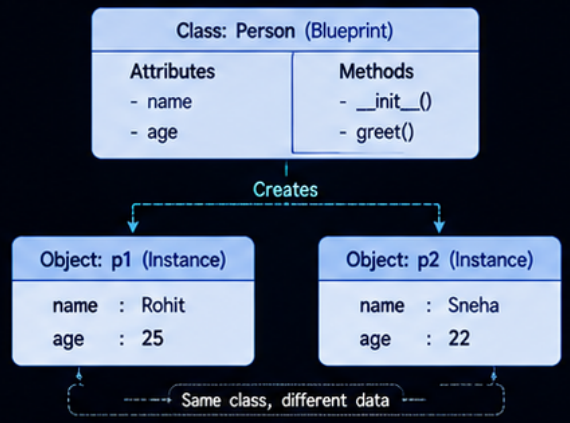
EXAMPLE: CREATING A CLASS & OBJECTS

```
1  class Person:
2      def __init__(self, name, age):
3          self.name = name
4          self.age = age
5
6      def greet(self):
7          print(f"Hi, I am {self.name}")
8          print(f"I am {self.age} years old.")
9
10 # Create objects (instances)
11 p1 = Person("Rohit", 25)
12 p2 = Person("Sneha", 22)
13
14 p1.greet()
15 p2.greet()
```

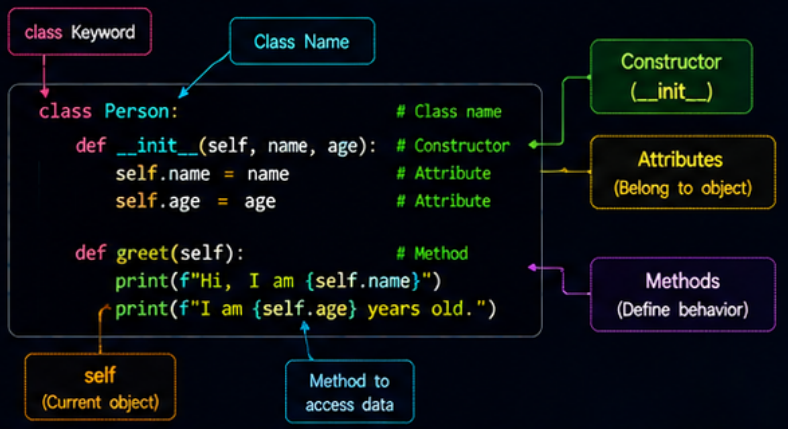
OUTPUT

```
Hi, I am Rohit
I am 25 years old.

Hi, I am Sneha
I am 22 years old.
```

VISUAL REPRESENTATION



ANATOMY OF A CLASS



KEY POINTS

- Class is a blueprint.
- Object is an instance of a class.
- __init__() is the constructor.
- self refers to the current object.
- Attributes store data.
- Methods define behavior.

MINI EXAMPLE

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

b = Book("Python 101", "CodeSardar")
print(b.title)    # Python 101
print(b.author)   # CodeSardar
```

COMMON OPERATIONS

Create object →	obj = ClassName()
Access attribute →	obj.attr
Call method →	obj.method()
Modify attribute →	obj.attr = value
Delete object →	del obj

REAL-WORLD EXAMPLES

- Car → Car class, objects are cars
- Student → Student class, objects are students
- Employee → Employee class, objects are employees
- Bank Account → Account class, objects are accounts

PRO TIP

OOP makes your code clean, reusable, modular and easy to maintain!

CONSTRUCTORS IN PYTHON

Initialize objects the right way!

Automatic Call	Initialize Data	Clean & Safe	Reusable	Best Practice
Called when object is created	Set initial values for object	Ensures object is in a valid state	Works every time an object is created	Use __init__() for better code

WHAT IS A CONSTRUCTOR?

A constructor is a special method __init__() that is automatically called when an object of a class is created. It is used to initialize the object's attributes.

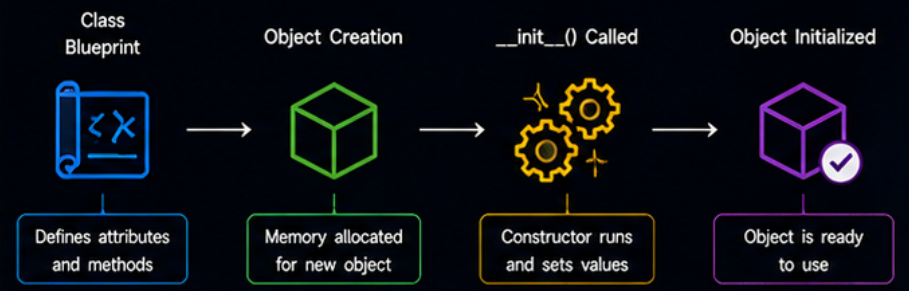
Constructor name is always __init__() (two underscores before and after).

CONSTRUCTOR SYNTAX

```
class ClassName:
    def __init__(self, parameters):
        # initialization code
        self.attribute = value
```

self	Parameters	Attributes	Values
Refers to the current object	Optional input values	Variables that belong to object	Initial values assigned

OBJECT CREATION FLOW



EXAMPLE: BASIC CONSTRUCTOR

```
1  class Student:
2      def __init__(self, name, age):
3          self.name = name      # attribute
4          self.age = age        # attribute
5
6  # Create object
7  s1 = Student("Rohit", 20)
8  print(s1.name)   # Rohit
9  print(s1.age)    # 20
```

OUTPUT

```
Rohit
20
```

__init__() is called automatically when s1 object is created.

CONSTRUCTOR WITH DEFAULT VALUES

```
1  class Person:
2      def __init__(self, name, age=18, city="Unknown"):
3          self.name = name
4          self.age = age
5          self.city = city
6
7  # Create objects
8  p1 = Person("Sneha")
9  p2 = Person("Aman", 25)
10 p3 = Person("Karan", 30, "Delhi")
11
12 print(p1.name, p1.age, p1.city)   # Sneha 18 Unknown
13 print(p2.name, p2.age, p2.city)   # Aman 25 Unknown
14 print(p3.name, p3.age, p3.city)   # Karan 30 Delhi
```

OUTPUT

```
Sneha 18 Unknown
Aman 25 Unknown
Karan 30 Delhi
```

Default values are used if arguments are not provided.

KEY POINTS ABOUT CONSTRUCTORS

- Constructor method name is __init__().
- It is automatically called when object is created.
- self is the reference to the current object.
- Used to initialize attributes of the object.
- Can take zero or more parameters.
- Can have default values for parameters.
- Helps in creating clean and reliable objects.

EXAMPLE OF __init__()

```
def __init__(self, param1, param2=default):
```

self	param1	param2	default
Current object	Required parameter	Optional parameter	Default value

WHEN IS __init__() CALLED?

1. When object is created.
2. Only once per object.
3. Before any other method is used.

COMMON MISTAKES

- Forgetting self parameter
- Using wrong name instead of __init__()
- Not initializing important attributes
- Too much logic in constructor

PRO TIP

Keep constructor simple.
Only initialize attributes.
Business logic should be in methods.

QUICK EXAMPLE (NO PARAMETERS)

```
1  class A:
2      def __init__(self):
3          self.x = 100
4
5  a = A()
6  print(a.x)   # 100
```

OUTPUT

```
100
```

Constructor = __init__() | self = Current Object | Initialize = Set Values | Object = Ready to Use | Good Code = Strong Foundation

INHERITANCE IN PYTHON

Reuse code. Extend functionality. Build powerful hierarchies!

Reuse Code	Extend Features	Method Override	Multiple Inheritance	Better Organization
Avoid repetition	Add new behavior	Change behavior	Inherit from many	Clean & modular code

WHAT IS INHERITANCE?

Inheritance allows a new class (child/subclass) to inherit attributes and methods from an existing class (parent/superclass).

Parent Class (Base Class) → Inheritance → Child Class (Derived Class)

inherits properties & methods

The child class can reuse, modify or extend the features of the parent class.

BASIC SYNTAX

```
class ParentClass:
    def parent_method(self):
        print("I am from Parent Class")

class ChildClass(ParentClass):
    def child_method(self):
        print("I am from Child Class")
```

- Parent (Base Class)
- Child (Derived Class)
- Inherits from Parent Class
- Can use Parent methods

HOW INHERITANCE WORKS

Parent Class (Base Class)
- Attributes
- Methods

Inheritance →

Child Class (Derived Class)
- Own Attributes
- Own Methods
- Inherited Methods

Creates Object → Child Class Object

Child class gets all public & protected members of parent class. It can add new members or override existing ones.

EXAMPLE: SINGLE INHERITANCE

```
1  class Animal:
2      def eat(self):
3          print("Animal is eating")
4
5  class Dog(Animal):
6      def bark(self):
7          print("Dog is barking")
8
9  d = Dog()
10 d.eat()   # Inherited method
11 d.bark()  # Own method
```

OUTPUT

```
Animal is eating
Dog is barking
```

Dog class inherited eat() from Animal class.

METHOD OVERRIDE

```
1  class Animal:
2      def sound(self):
3          print("Animal makes a sound")
4
5  class Cat(Animal):
6      def sound(self):    # Overriding method
7          print("Cat says: Meow")
8
9  c = Cat()
10 c.sound()
```

OUTPUT

```
Cat says: Meow
```

Child class can change the behavior of parent class methods.

USING super()

```
1  class Parent:
2      def __init__(self, name):
3          self.name = name
4      def show(self):
5          print(f"Name: {self.name}")
6
7  class Child(Parent):
8      def __init__(self, name, age):
9          super().__init__(name)   # Call parent constructor
0          self.age = age
10     def show(self):
12         super().show()          # Call parent method
13         print(f"Age: {self.age}")
```

OUTPUT

```
Name: Rohit
Age: 20
```

super() helps to access parent class methods & constructor.

MULTIPLE INHERITANCE

```
1  class Father:
2      def skills(self):
3          print("Driving")
4  class Mother:
5      def skills(self):
6          print("Cooking")
7  class Child(Father, Mother):
8      pass
9  c = Child()
10 c.skills()  # Output: Driving (first parent)
```

Father (Base Class) + Mother (Base Class) → Child (Derived Class)

Child class can inherit from multiple parent classes.

Python follows MRO (Method Resolution Order).

KEY BENEFITS

- Code Reusability – Reuse existing class features.
- Extensibility – Add new features easily.
- Maintainability – Easy to update and maintain code.
- Modularity – Organize code in a logical way.
- Flexibility – Supports multiple inheritance.

PRO TIP
Always prefer composition over inheritance when possible. Use inheritance only when "is-a" relationship exists.

TERMS YOU SHOULD KNOW

Parent Class / Base Class	The class whose members are inherited.
Child Class / Derived Class	The class that inherits from parent class.
super() Function	Used to access methods of parent class.
Method Override	Redefining a method in child class.
Multiple Inheritance	Inheritance from more than one parent.

Base Class = Parent Class | Derived Class = Child Class | super() = Access Parent | Override = Change Behavior | Multiple = More Power | Code Smart, Code Reuse!

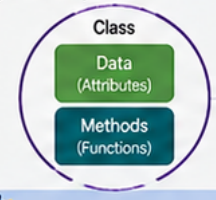
ENCAPSULATION IN PYTHON

Wrap data and code together. Restrict direct access to some details.

Data Hiding	Better Control	Reusability	Flexibility	Maintainability
Protect data from direct access	Validate and control access to data	Easier to reuse and maintain	Change internals without affecting code	Clean, modular and organized

WHAT IS ENCAPSULATION?

Encapsulation is the concept of bundling data (attributes) and methods together in a class and restricting direct access to some of the object's components.



Encapsulation protects the internal state of an object.

Access data through methods, not directly.

ACCESS MODIFIERS (BY CONVENTION)

Public (No underscore)

- Accessible from anywhere
- Default access level

```
class Example:
    def __init__(self):
        self.name = "Rohit"  # public
```

```
obj = Example()
print(obj.name)  # ✅ Allowed
```

Protected (Single underscore _)

- Intended for internal use
- Accessible within class and subclasses

```
class Example:
    def __init__(self):
        self._age = 25  # protected
```

```
obj = Example()
print(obj._age) # ⚠️ Allowed (not recommended)
```

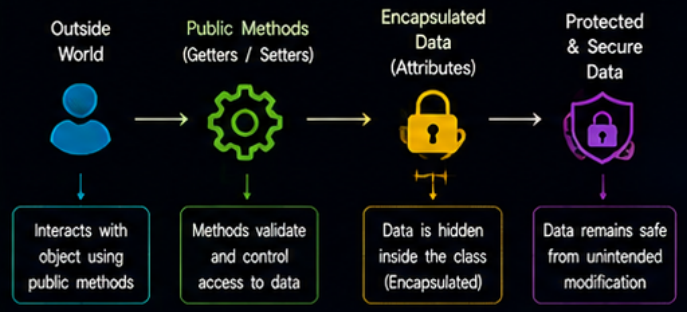
Private (Double underscore __)

- Not accessible directly
- Name mangling is applied
- More secure

```
class Example:
    def __init__(self):
        self.__salary = 50000  # private
```

```
obj = Example()
print(obj.__salary)  # ❌ Error
```

ENCAPSULATION FLOW


Outside World
Public Methods (Getters / Setters)
Encapsulated Data (Attributes)
Protected & Secure Data
Interacts with object using public methods
Methods validate and control access to data
Data is hidden inside the class (Encapsulated)
Data remains safe from unintended modification

GETTERS AND SETTERS

Use getters and setters to access and modify private data safely.

Example: Using Getters & Setters

```
class Person:
    def __init__(self, name, age):
        self.__age = age       # private
        self.name = name       # public

    def get_age(self):
        return self.__age

    def set_age(self, age):
        if age >= 0:
            self.__age = age
        else:
            print("Age cannot be negative!")

p = Person("Rohit", 20)
print(p.get_age())  # 20
p.set_age(25)
print(p.get_age())  # 25
p.set_age(-5)       # Age cannot be negative!
```

Why Use Getters & Setters?

- Validate data before setting
- Keep data safe and consistent
- Control how data is accessed
- Add logic later without changing external code

Access Summary

Modifier	Access From
Public (no _)	Everywhere
Protected (_)	Class, Subclasses (not recommended outside)
Private (__)	Class only (name mangled)

NAME MANGLING (PRIVATE VARIABLES)

Python changes the name of private variables to prevent direct access.

```
class Demo:
    def __init__(self):
        self.__secret = "TopSecret"
```

How it works?

```
obj = Demo()
print(obj.__secret)          # ❌ AttributeError
print(obj._Demo__secret)     # ✅ Allowed (Name Mangling)
```

Name mangling makes it harder (not impossible) to access private data.

EXAMPLE: ENCAPSULATION IN ACTION

```
class BankAccount:
    def __init__(self, owner, balance):
        self.__balance = balance   # private
        self.owner = owner         # public

    def deposit(self), amount):
        if amount > 0:
            self.__balance += amount
            print("Deposited:", amount)
        else:
            print("Invalid amount!")

    def get_balance(self):
        return self.__balance

a = BankAccount("Rohit", 1000)
a.deposit(500)
print(a.get_balance())    # 1500
# print(a.__balance)      # ❌ Error
```

Key Takeaway

Data is hidden inside the class and accessed only through methods.

Safe & Secure

BEST PRACTICES

- Use encapsulation to protect data.
- Use getters and setters for controlled access.
- Use _ (protected) for internal use.
- Use __ (private) for sensitive data.
- Keep data validation inside setters.
- Encapsulation makes code more robust and flexible.

REMEMBER

Encapsulation is about controlling access, not completely blocking it. It's a convention in Python, not a strict rule like in other languages.

QUICK REFERENCE

self.var	→ Public
self._var	→ Protected
self.__var	→ Private (Name Mangled)
self.__var	→ Access via methods

BENEFITS OF ENCAPSULATION

- Protects data from accidental changes
- Provides better control over data
- Improves code maintainability
- Makes code more secure and reliable

REAL-WORLD EXAMPLE

Think of a car 🚗
You drive (use public methods)
But the engine (data) is hidden inside and protected from direct access.

PRO TIP

Always use meaningful method names for get_*, set_* to make your code clean and self-explanatory.

SUMMARY

Encapsulation = Hiding data + Controlling access
Use it wisely to build safe, strong and scalable applications!

POLYMORPHISM IN PYTHON

One Interface, Multiple Forms!

Same Interface	Different Behavior	Method Override	Duck Typing	OOP Principle
Use same method name	Each object behaves differently	Child class redefines parent method	If it walks like a duck and quacks like a duck...	Core concept of object-oriented programming

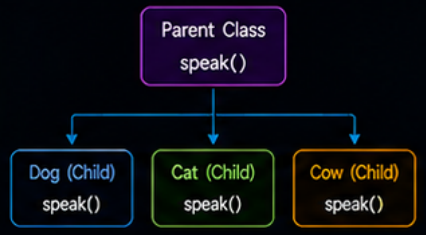
WHAT IS POLYMORPHISM?

Polymorphism means "many forms". In Python, it allows us to use a single interface (method name) to perform different tasks based on the object.

One Method (Same Interface) → Many Behaviors (Different Implementation)

Polymorphism makes code flexible, reusable and easy to extend.

POLYMORPHISM USING METHOD OVERRIDE




```
class Animal:
    def speak(self):
        pass

class Dog(Animal):
    def speak(self):
        return "Woof!"
class Cat(Animal):
    def speak(self):
        return "Meow!"
class Cow(Animal):
    def speak(self):
        return "Moo!"
```

OUTPUT

```
Woof!
Meow!
Moo!
```

Each child class provides its own implementation of the same behavior changes depending on the object.

POLYMORPHISM IN ACTION

```
def animal_sound(animal):
    print(animal.speak())

dog = Dog()
cat = Cat()
cow = Cow()

animal_sound(dog)
animal_sound(cat)
animal_sound(cow)
```

OUTPUT

```
Woof!
Meow!
Moo!
```

Same function (animal_sound) works for different objects. That's Polymorphism!

POLYMORPHISM WITH DUCK TYPING

Python uses Duck Typing – If it walks like a duck and quacks like a duck, then it's a duck!

```
def make_sound(obj):
    obj.sound()

class Duck:
    def sound(self):
        return "Quack!"
class Dog:
    def sound(self):
        return "Bark!"

d = Duck()
dog = Dog()

make_sound(d)
make_sound(dog)
```

OUTPUT

```
Quack!
Bark!
```

Objects don't need to be from the same class hierarchy. If they have the same method, they can be used.

COMPARISON: METHOD OVERRIDE vs DUCK TYPING

Feature	Method Override	Duck Typing
Based On	Inheritance	Behavior (not inheritance)
Requirement	Requires parent-child relationship	Only same method name is required
Checked By	Class relationship	Python runtime
Flexibility	Less flexible	More flexible
Example	Dog, Cat inherit from Animal	Duck and Dog have sound()

REAL-WORLD EXAMPLE

Different vehicles have the same method: start()

```
class Car:
    def start(self):
        return "Car started"

class Bike:
    def start(self):
        return "Bike started"

def start_vehicle(vehicle):
    print(vehicle.start())

start_vehicle(Car())
start_vehicle(Bike())
```

OUTPUT

```
Car started
Bike started
```

Same function start_vehicle() works for Car and Bike objects. Polymorphism!

KEY BENEFITS

- Code reusability
- Easy to extend
- Improves readability
- Supports scalability
- Reduces complexity

POLYMORPHISM FLOW

Same Method (Interface) → Different Objects (Child Classes) → Different Behavior (Implementation) → Polymorphic Result

PRO TIP

Always code to an interface, not an implementation. Let objects decide how they behave!

</>

QUICK REFERENCE

- Same method name
- Different implementations
- Method Override (inheritance)
- Duck Typing (behavior based)
- Polymorphism = Flexibility

SUMMARY

Polymorphism allows different classes to be treated as instances of the same class through a common interface. It's one of the most powerful features of OOP in Python.

ABSTRACTION IN PYTHON

Focus on what an object does, not how it does it.

Hide Details	Define Structure	Easy to Extend	Reduce Complexity	Better Design
Hide complex implementation	Define what must be done	New features in child classes	Focus on important things	Clean, modular and maintainable

WHAT IS ABSTRACTION?

Abstraction is the concept of hiding the internal implementation and showing only the essential features of an object.

- Show only what is necessary.
- Hide how it is done.
- User interacts with the interface, not the internal logic.

Abstraction helps reduce complexity and improve code reusability.

ABSTRACTION SYNTAX

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass
```

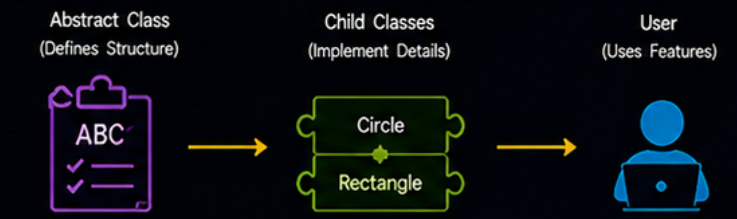
Abstract class cannot be instantiated. It provides a blueprint for child classes.

ABC — Base class for creating abstract classes

abstractmethod — Decorator to declare abstract methods

Abstract Class — Cannot be instantiated directly

HOW ABSTRACTION WORKS



Define methods (abstractmethod) No Implementation

Provide specific implementation for abstract methods

Interact using common interface (Shape)

EXAMPLE: SHAPE AREA

```
1  from abc import ABC, abstractmethod
2
3  class Shape(ABC):
4      @abstractmethod
5      def area(self):
6          pass
7
8  class Circle(Shape):
9      def __init__(self, r):
10         self.r = r
10     def area(self):
11         return 3.1416 * self.r * self.r
12
14 class Rectangle(Shape):
15     def __init__(self, l, b):
16         self.l = l
17         self.b = b
18     def area(self):
19         return self.l * self.b
20
21 # Usage
22 c = Circle(5)
23 r = Rectangle(4, 6)
24 print("Circle Area:", c.area())    # 78.54
25 print("Rectangle Area:", r.area()) # 24
```

OUTPUT

```
Circle Area: 78.54
Rectangle Area: 24
```

KEY TAKEAWAY

Shape defines what (area must exist), but Circle and Rectangle decide how to calculate it.

KEY BENEFITS

Hide Implementation
Users see only the necessary features, not the internal code.

Define Standard
Abstract classes define a standard for child classes to follow.

Easy to Extend
Add new child classes without changing existing code.

Improve Maintainability
Code becomes clean, organized and easy to maintain.

ABSTRACTION IN REAL LIFE

Car	You drive the car (what it does), not how the engine works (how it does).
ATM	You enter PIN and get cash (what), not how the machine processes it (how).
Smartphone	You use apps and features (what), not how the hardware is managed (how).

QUICK REFERENCE

ABC	Base class for abstract classes
abstractmethod	Used to declare abstract methods
Abstract Class	Blueprint for other classes
Interface	Defines a contract (Method signatures)
Hide Details	Hide internal implementation
OOP Principle	Abstraction is one of the 4 pillars of OOP

Abstraction = Simplicity + Clarity + Reusability

Good Code is not about writing more, it's about hiding the unnecessary.

Write clean code. Think smart. Use Abstraction!

MAGIC METHODS IN PYTHON

Special methods with double underscores (dunder methods)
Let your objects behave like built-in types! ✨

Magic Happens	Built-in Like	Readable Code	More Power	Best Practice
Automatic calls by Python	Make your objects easy to use	Cleaner and more pythonic	Do more with less code	Follow Python conventions

WHAT ARE MAGIC METHODS?

Magic methods (or dunder methods) are special methods in Python classes that have double underscores at the start and end of their names. They are automatically called by Python in certain situations.

`__method__`

They allow you to define behavior for built-in operations on your custom objects.

HOW IT WORKS

1. You define magic methods in your class.
2. Python calls them automatically.
3. Your object behaves like built-in types.

You don't call these methods directly, Python does it for you!

MOST COMMON MAGIC METHODS

__init__()
Constructor
Called when object is created.

```
def __init__(self, name):
    self.name = name
```

```
obj = MyClass("Rohit")
```

__str__()
String Representation
Called by str() and print().

```
def __str__(self):
    return f"Name: {self.name}"
```

```
print(obj)  # Name: Rohit
```

__repr__()
Official Representation
Called by repr() and in debug.

```
def __repr__(self):
    return f"MyClass({self.name})"
```

```
repr(obj)  # MyClass('Rohit')
```

__len__()
Length
Called by len().

```
def __len__(self):
    return len(self.items)
```

```
len(obj)  # 5
```

__eq__()
Equality
Called by == operator.

```
def __eq__(self, other):
    return self.id == other.id
```

```
obj1 == obj2  # True / False
```

__add__()
Addition
Called by + operator.

```
def __add__(self, other):
    return self.value + other.value
```

```
obj1 + obj2  # new object
```

__getitem__()
Indexing / Slicing
Called by obj(key).

```
def __getitem__(self, key):
    return self.data[key]
```

```
obj[0]  # first item
```

__call__()
Callable Object
Called when object is called like a function.

```
def __call__(self, *args, **kwargs):
    return self.run(*args, **kwargs)
```

```
obj(10)  # call object
```

EXAMPLE: SIMPLE CLASS USING MAGIC METHODS

```
1  class Book:
2      def __init__(self, title, pages):
3          self.title = title
4          self.pages = pages
5
6      def __str__(self):
7          return f"{self.title} ({self.pages} pages)"
8
9      def __len__(self):
10         return self.pages
```

USAGE & OUTPUT

```
b = Book("Python 101", 350)
print(b)
len(b)
```

OUTPUT

```
Python 101 (350 pages)
350
```

WHY USE MAGIC METHODS?

- Make your classes easy to use.
- Improve readability and developer experience.
- Work smoothly with Python built-in functions.
- Follow Pythonic way of coding.

Remember: Magic methods make your objects smart and Pythonic!

Tip: Use only what you need. Don't overload unnecessarily.

Goal: Clean, readable and maintainable code!

DECORATORS IN PYTHON

Add extra behavior to functions or methods.

Clean • Reusable • Powerful ★

Simple Syntax	Reusable Logic	Clean Code	Enhance Functionality	Built-in Support
Just use @ above function	Write once, use anywhere	Better separation of concerns	Extend behavior without modifying code	Python has many decorators

WHAT IS A DECORATOR?

A decorator is a function that takes another function and extends its behavior without permanently modifying it.

Function (Original) → @ Decorator → Enhanced Function

Decorators help you add functionality like logging, timing, authentication, caching, etc.

HOW DECORATOR WORKS

Function (Original) → Decorator (@decorator) → Enhanced Function

Wrapper Function (Inner Function)

BEFORE — Extra logic runs before the function executes.

AFTER — Extra logic runs after the function executes.

BASIC SYNTAX

```
def decorator_func(original_func):
    def wrapper_func(*args, **kwargs):
        # Before
        print("Before function call")
        result = original_func(*args, **kwargs)
        # After
        print("After function call")
        return result
    return wrapper_func
```

```
@decorator_func
def my_function():
    print("Hello from function!")
```

- @decorator — Apply decorator
- wrapper_func — Wrapper function that adds logic
- original_func — Original function (The one being decorated)
- return — Always return the wrapper

EXAMPLE: SIMPLE DECORATOR

```
def my_decorator(func):
    def wrapper(*args, **kwargs):
        print("[BEFORE] Function is running")
        result = func(*args, **kwargs)
        print("[AFTER] Function has finished")
        return result
    return wrapper

@my_decorator
def greet(name):
    print(f"Hello, {name}!")

greet("Rohit")
```

OUTPUT

```
[BEFORE] Function is running
Hello, Rohit!
[AFTER] Function has finished
```

★ The decorator adds behavior before and after the original function call.

DECORATOR WITH PARAMETERS

```
def repeat(times):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for i in range(times):
                print(f"--- Run {i+1} ---")
                func(*args, **kwargs)
        return wrapper
    return decorator

@repeat(3)
def say_hello():
    print("Hello, Decorators!")

say_hello()
```

OUTPUT

```
--- Run 1 ---
Hello, Decorators!
--- Run 2 ---
Hello, Decorators!
--- Run 3 ---
Hello, Decorators!
```

DECORATOR WITH CLASS

```
class MyDecorator:
    def __init__(self, func):
        self.func = func

    def __call__(self, *args, **kwargs):
        print("[BEFORE]")
        result = self.func(*args, **kwargs)
        print("[AFTER]")
        return result

@MyDecorator
def add(a, b):
    print(f"Sum: {a + b}")

add(5, 7)
```

OUTPUT

```
[BEFORE]
Sum: 12
[AFTER]
```

COMMON USE CASES

- **Logging** — Log function calls and details
- **Timing** — Measure execution time
- **Authentication** — Check access before running
- **Caching** — Store results for faster access
- **Input Validation** — Validate arguments

QUICK REFERENCE

Use @decorator above the function.

- ✓ Decorators wrap functions and extend their behavior.
- ✓ Always return the wrapper function.
- ✓ Can be used with or without parameters.

DECORATOR FLOW

Function (Original) → Decorator (@decorator) → Wrapper Function (Adds Logic) → Enhanced Function (Runs with extra behavior)

PRO TIP

Stack multiple decorators to add more than one behavior!

```
@decorator_one
@decorator_two
def my_function():
    pass
```

REMEMBER

Decorators = Reusable + Clean + Powerful

- ✓ Make your code modular and easy to maintain.
- ✓ Python's built-in decorators: @property, @staticmethod, @classmethod, @lru_cache, etc.

GENERATORS IN PYTHON

Generate values on the fly, one at a time!

Lazy Evaluation	Memory Efficient	Iterator	Sequence	Reusable	Powerful
Compute only when needed	Don't store all values in memory	Works with iter() and next()	Behaves like a sequence of values	Use in loops, functions & more	Great for large data & streams

WHAT IS A GENERATOR?

A generator is a special type of iterator that generates values one at a time using the `yield` keyword.

```
def func():
    yield 1
    yield 2
    ...
```

→ Generator Object

Generators are created using functions with `yield` instead of return.

HOW GENERATOR WORKS

Generator Function (with `yield`) → Generator Object (Iterator) → `next()` Fetch next value → `StopIteration` (End of sequence)

```
def count_up_to(n):
    i = 1
    while i <= n:
        yield i    # pause & return value
        i += 1     # resume from here
```

```
Output:
1
2
3
...
n
```

KEY WORDS EXPLAINED

`yield`	Pause the function and return a value. State of function is saved for next call.
`yield from`	Delegate to another generator.
`next()`	Fetch the next value from the generator.
Generatior	Function with `yield` that returns an iterator.
Iterator	Object that can be iterated using `next()`.
Lazy Evaluation	Values are generated only when requested, not all at once.

EXAMPLE: SIMPLE GENERATOR

```
1  def count(n):
2      i = 1
3      while i <= n:
4          yield i
5          i += 1
6
7  g = count(5)
8  print(next(g))   # 1
9  print(next(g))   # 2
10 print(next(g))   # 3
11 for x in g:      # 4, 5
12     print(x)
```

OUTPUT

```
1
2
3
4
5
```

The function remembers where it left off!

YIELD FROM EXAMPLE

```
1  def sub_gen():
2      yield from [10, 20, 30]
3
4  def main_gen():
5      print("Start")
6      yield from sub_gen()
7      print("End")
8
9  g = main_gen()
10 for val in g:
11     print(val)
```

OUTPUT

```
Start
10
20
30
End
```

`yield from` simplifies working with other generators.

GENERATORS vs LISTS

List (Eager Evaluation)		Generator (Lazy Evaluation)
✗ Loads all items in memory	Memory	Uses memory efficiently ✓
✗ Slower for large data	Speed	Faster, generates on demand ✓
✗ Small data, random access	Best for	Large data, streams, iteration ✓
`[x for x in range(1000000)]`	Example	`(x for x in range(1000000))`

COMMON USE CASES

- Handling large files
- Data streaming
- Infinite sequences
- Pipelines & processing
- Real-time data feeds

Perfect when data is huge or infinite!

GENERATOR FLOW

Define function with `yield` → Create generator object → `next()` Get next value → Repeat until StopIteration

PRO TIP

Use generators to save memory and process large data smoothly.

```
# Instead of this (high memory)
my_list = [x*x for x in range(10**7)]
# Use this (low memory)
my_gen = (x*x for x in range(10**7))
```

BUILT-IN FUNCTIONS THAT WORK

`next()`	`iter()`	`list()`	`sum()`	`any()`, `all()`
Get next value	Get iterator object	Convert to list	Sum values	Check conditions

REMEMBER

- Generators use yield, not return.
- They are iterators.
- They generate values one at a time.
- Memory efficient & fast.
- Use when working with large data!

ITERATORS IN PYTHON

Iterate over data, one value at a time!

- **Memory Efficient** — Don't load all data in memory
- **Efficient Looping** — Perfect for large datasets
- **Works Everywhere** — Lists, Tuples, Strings, Files, Generators...
- **Built-in Protocol** — iter() & next() do the magic
- **Pythonic** — The foundation of Python loops

WHAT IS AN ITERATOR?

An iterator is an object that allows you to iterate (loop) over a sequence of values one at a time.

Iterable (Source) → iter() → Iterator (Object) → next() → Value 1 | Value 2 | Value 3 | ... | StopIteration

Iterators remember their state, so each call to next() gives the next value.

HOW ITERATORS WORKS

Iterable (Source) → Get iterator object → next() Get next value → StopIteration No more values

```
Example Flow:
my_list = [10, 20, 30]
it = iter(my_list)

next(it)   # 10
next(it)   # 20
next(it)   # 30
next(it)   # StopIteration (no more items)
```

Raises StopIteration when no more values.

ITERATOR PROTOCOL

__iter__() — Returns the iterator object (itself)

__next__() — Returns the next value. Raises StopIteration when done.

```
class MyIterator:
    def __iter__(self):
        return self       # Return iterator object (itself)

    def __next__(self):
        # Return next value or raise StopIteration
        pass
```

Every iterator must implement __iter__() and __next__().

EXAMPLE: SIMPLE ITERATOR

```
1  class CountDown:
2      def __init__(self, start):
3          self.current = start
4
5      def __iter__(self):
6          return self
7
8      def __next__(self):
9          if self.current <= 0:
10             raise StopIteration
11         value = self.current
12         self.current -= 1
14         return value
```

USAGE

```
cd = CountDown(3)
for num in cd:
    print(num)
```

OUTPUT

```
3
2
1
```

ITERABLE vs ITERATOR

Iterable	Iterator
Object that can return an iterator.	Object that represents a stream of data.
Has __iter__() method.	Has __iter__() and __next__().
Can be used in loops.	Keeps track of its state.
Examples: list, tuple, str, dict, set, file, etc.	Calls next() to get values one by one.
Check: hasattr(obj, '__iter__')	Check: hasattr(obj, '__next__')

BEHIND THE SCENES (FOR LOOP)

```
my_list = [100, 200, 300]
it = iter(my_list)          # Get iterator

while True:
    try:
        value = next(it)      # Get next value
        print(value)
    except StopIteration:  # No more items
        break
```

A for loop automatically handles StopIteration for you.

COMMON ITERABLES

list	`[1, 2, 3]`
tuple	`(1, 2, 3)`
string	`"abc"`
dict	`{'a': 1, 'b': 2}`
set	`{1, 2, 3}`
file	`open('file.txt')`
range	`range(5)`
generator	`(x for x in range(3))`

ITERATOR FLOW

Iterable (Object) → iter() → next() → ... → StopIteration (End)

KEY POINTS

- iter() gets the iterator from an iterable.
- next() gets the next value.
- StopIteration is raised when done.
- Iterators are one-time use.

PRO TIP

Use iter() and next() when you need more control over iteration, especially in advanced use cases.

```
it = iter(data)
while True:
    try:
        print(next(it))
    except StopIteration:
        break
```

REMEMBER

- Iterable → Can be iterated.
- Iterator → Does the iteration.
- iter() → Get iterator.
- next() → Get next value.
- StopIteration → End of data.

PYTHONIC LOOP (RECOMMENDED)

```
for item in iterable:
    # Python handles everything
    print(item)
```

REGULAR EXPRESSIONS IN PYTHON

Powerful pattern matching for Text Search & Manipulation

Text Search	Replace	Split	Extract	Powerful
Find patterns in text	Find & replace text easily	Split text by pattern	Get useful data from text	Fast, flexible & very useful

WHAT IS REGEX?

Regular Expressions (Regex) are patterns used to match, search, and manipulate text.

re → Module → Pattern Matching

Python provides the re module to work with regular expressions.

REGEX WORKFLOW

Input Text → re (Regex Pattern) → Match / Search Engine → Result (Matches)

```
import re
pattern = r"\d+"    # pattern (one or more digits)
text = "Order 1234 is placed on 2024-05-19"
matches = re.findall(pattern, text)  # ['1234', '2024', '05', '19']
```

COMMON FUNCTIONS IN re MODULE

Function	Description	Example
match()	Checks for a match only at the beginning of the text.	`re.match(r'\d+', '123abc')` `# Match object`
search()	Searches for the first occurrence anywhere in the text.	`re.search(r'\d+', 'abc123')` `# Match object`
findall()	Returns a list of all non-overlapping matches.	`re.findall(r'\d+', 'a1b22c333')` `# ['1', '22', '333']`
sub()	Replaces matches with a specified string.	`re.sub(r'\d+', '#', 'a1b22c333')` `# 'a#b#c#'`
split()	Splits the text by the pattern.	`re.split(r',\s*', 'one, two, three')` `# ['one', 'two', 'three']`

COMMON PATTERNS (CHEAT SHEET)

Pattern	Matches
.	Any single character
^	Start of string
$	End of string
*	Zero or more of the previous
+	One or more of the previous
?	Zero or one of the previous
\d	Digit (0-9)
\D	Non-digit
\w	Word character (a-z, A-Z, 0-9, _)
\W	Non-word character
\s	Whitespace (space, tab, newline)
\S	Non-whitespace
[abc]	Any one of a, b or c
[^abc]	Any character except a, b or c
[a-z]	Any lowercase letter
( )	Capture group
|	OR operator

EXAMPLES

Use Case	Pattern	Example	Matches
Extract Emails	`\b\w+@[\w.-]+\.\w+\b`	test@mail.com	['test@mail.com']
Extract Numbers	`\d+`	abc 123 def 456	['123', '456']
Extract Words	`\b\w+\b`	Hello, World!	['Hello', 'World']
Phone Number	`\d{10}`	Call 9876543210 now	['9876543210']
Date (YYYY-MM-DD)	`\d{4}-\d{2}-\d{2}`	2024-05-19	['2024-05-19']
Remove Extra Spaces	`\s+` (use with sub)	a b c	['a', 'b', 'c']

SUB() EXAMPLE (REPLACE)

```
import re
text = "Order 1234, Price: 567.89"
new_text = re.sub(r'\d+', '#', text)
print(new_text)

Output:
Order #, Price: #.#
```

1234 → sub() → ####

SPLIT() EXAMPLE

```
import re
text = "apple, banana, cherry"
items = re.split(r',\s*', text)
print(items)

Output:
['apple', 'banana', 'cherry']
```

a, b, c → split() → a | b | c

RAW STRING (r"...")

```
# Use raw strings to avoid escaping backslashes
pattern = r"\d+\w+"    # Correct
# pattern = "\\d+\\w+" # Also works but needs escaping
```

Raw strings make regex easier to write!

FLAGS (OPTIONS)

re.IGNORECASE (re.I)	Case-insensitive matching
re.MULTILINE (re.M)	^ and $ work on each line
re.DOTALL (re.S)	. matches newline too
re.VERBOSE (re.X)	Allows comments & spaces

FULL WORKING EXAMPLE

```
import re
text = "User: john.doe@email.com,  Phone: 9876543210"
email = re.search(r'[\w.-]+@[\w.-]+', text)
phone = re.search(r'\d{10}', text)
print("Email:", email.group())     # john.doe@email.com
print("Phone:", phone.group())     # 9876543210
```

TIPS & BEST PRACTICES

- Always use raw strings (r"...").
- Test your regex using regex101.com.
- Keep patterns simple and readable.
- Use re.VERBOSE for complex patterns.
- Handle None when no match is found.

Regex = Find, Match, Replace, Split — All in One!

Master Regex, Master Text Processing!

Practice makes perfect. Keep experimenting!

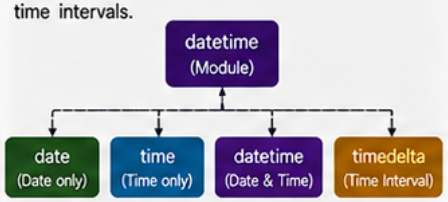
DATE & TIME IN PYTHON

Work with Dates, Times & Intervals Easily

Easy to Use	Accurate	Time Saving	Flexible	Everywhere
Simple & powerful date and time handling	Timezone aware and reliable results	Built-in functions save your development time	Parse, format, calculate & more with ease	Useful in logs, scheduling, reports and APIs

WHAT IS DATETIME?

Python has a datetime module to work with dates, times and time intervals.



The datetime module provides classes for manipulating dates and times in Python.

CORE CLASSES IN datetime MODULE

date
Represents a date (year, month, day)

```
from datetime import date

d = date(2024, 5, 19)
print(d)          # 2024-05-19
print(d.year)     # 2024
print(d.month)    # 5
print(d.day)      # 19
```

time
Represents a time (hour, minute, second)

```
from datetime import time

t = time(14, 30, 45)
print(t)          # 14:30:45
print(t.hour)     # 14
print(t.minute)   # 30
print(t.second)   # 45
```

datetime
Represents both date and time

```
from datetime import datetime

dt = datetime(2024, 5, 19, 14, 30, 45)
print(dt)
# 2024-05-19 14:30:45
print(dt.date())  # 2024-05-19
print(dt.time())  # 14:30:45
```

timedelta
Represents a time difference / duration

```
from datetime import timedelta

delta = timedelta(days=2, hours=5)
                  minutes=30)
print(delta)
# 2 days, 5:30:00
```

COMMON METHODS

now() Current date & time	`datetime.now()` `# 2024-05-19 14:30:45.123456`
today() Current date (date class)	`date.today()` `# 2024-05-19`
strftime() Format date/time as string	`dt.strftime("%Y-%m-%d %H:%M:%S")` `# '2024-05-19 14:30:45'`
strptime() Parse string to date/time	`datetime.strptime("2024-05-19", '%Y-%m-%d")` `# returns datetime object`
timestamp() Convert to timestamp	`dt.timestamp()` `# 1716119445.123456`

DATE & TIME EXAMPLES

```
from datetime import date, time, datetime, timedelta

# Today's date
today = date.today()
print("Today:", today)       # 2024-05-19

# Current date & time
now = datetime.now()
print("Now:", now)           # 2024-05-19 14:30:45.123456

# Create specific datetime
dt = datetime(2024, 5, 19, 14, 30, 45)
print("Custom:", dt)

# Time part
tm = time(9, 15, 0)
print("Time:", tm)

# Combine date & time
dt2 = datetime.combine(today, tm)
print("Combine:", dt2)
```

TIMEDELTA EXAMPLES

```
from datetime import datetime, timedelta
now = datetime.now()

# Add time
after_2_days = now + timedelta(days=2)
print("After 2 days:", after_2_days)

# Subtract time
before_3_hours = now - timedelta(hours=3)
print("3 hours before:", before_3_hours)

# Difference between two datetimes
dt1 = datetime(2024, 5, 19, 10, 0, 0)
dt2 = datetime(2024, 5, 20, 12, 30, 0)
diff = dt2 - dt1
print("Difference:", diff)     # 1 day, 2:30:00
print("Total seconds:", diff.total_seconds())
```

STRFTIME() FORMAT CODES

Code	Meaning	Code	Meaning
%Y	4-digit year (2024)	%H	Hour (00-23)
%y	2-digit year (24)	%I	Hour (01-12)
%m	Month (01-12)	%M	Minute (00-59)
%B	Month name (May)	%S	Second (00-59)
%d	Day (01-31)	%f	Microsecond
%A	Weekday name (Sunday)	%p	AM or PM
%a	Weekday short (Sun)	%Z	Timezone name

```
Example: dt.strftime("%A, %d %B %Y %I:%M:%S %p")
# Sunday, 19 May 2024 02:30:45 PM
```

STRPTIME() EXAMPLES (STRING TO DATETIME)

```
from datetime import datetime

# Parse string to datetime
dt = datetime.strptime("19-05-2024 14:30:45", "%d-%m-%Y %H:%M:%S")
print(dt)      # 2024-05-19 14:30:45

# Parse date only
d = datetime.strptime("19/05/2024", "%d/%m/%Y").date()
print(d)       # 2024-05-19

# Parse time only
t = datetime.strptime("02:30 PM", "%I:%M %p").time()
print(t)       # 14:30:00
```

PRO TIP

Use `datetime.now()` for current date & time and `strefime()` for formatting before displaying or saving.

BEST PRACTICES

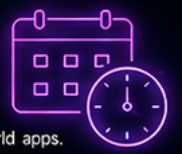
- Always be aware of timezone while working with datetime.
- Store dates in ISO format: YYYY-MM-DD.
- Use UTC time for consistent data across systems.
- Use timedelta for date calculations instead of manual math.

QUICK REFERENCE

Task	Use
Current date & time	`datetime.now()`
Current date	`date.today()`
Format datetime	`dt.strffime("format")`
Parse string to datetime	`datetime.strptime(str, "format")`
Time difference	`dt2 - dt1 (returns timedelta)`

NOTE

- Months are 1-12, not 0-11.
- Days are 1-31.
- Datetimes are immutable.
- Use timezone-aware objects for real-world apps.



RANDOM MODULE IN PYTHON

Generate Random Numbers, Selections & Sequences

- **Easy to Use** — Simple functions powerful results
- **Versatile** — Numbers, choices, shuffle & more
- **Reliable** — Useful for games, simulations & tests
- **Efficient** — Fast performance & optimized
- **Built-in** — Comes with Python by default

WHAT IS RANDOM MODULE?

The random module in Python is used to generate random numbers, make random selections, shuffle data and more.

```
import random

# Start using
random functions
```

Randomness makes programs more dynamic, unpredictable and fun!

RANDOM FUNCTIONS OVERVIEW

random()	randint(a, b)	choice(seq)	choices(seq, k)	shuffle(seq)	sample(pop, k)	randrange()
Random float between 0.0 and 1.0	Random integer N such that a ≤ N ≤ b	Random item from a sequence	Random items with replacement (k times)	Shuffle the sequence in place	Random unique items from population	Random number from range like range()

All functions are in the random module. Import it once and use anywhere!

QUICK EXAMPLE

```
import random

print("random():", random.random())
print("randint(1, 10):", random.randint(1, 10))
print("choice(['A','B','C']):", random.choice(['A','B','C']))
print("choices([1,2,3,4], k=3):", random.choices([1,2,3,4], k=3))

lst = [1, 2, 3, 4, 5]
random.shuffle(lst)
print("shuffle:", lst)
print("sample([10,20,30,40,50], 3):", random.sample([10,20,30,40,50], 3))
print("randrange(1, 10, 2):", random.randrange(1, 10, 2))
```

DETAILED FUNCTION REFERENCE

random()

Return a random float in the range [0.0, 1.0).

```
import random
print(random.random())
```

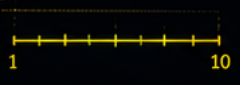
Output: `0.7312453879`

randint(a, b)

Return a random integer N such that a ≤ N ≤ b.

```
import random
print(random.randint(1, 10))
```

Output: `7`



choice(seq)

Return a random element from the non-empty sequence.

```
import random
print(random.choice(
  ('red','green','blue')))
```

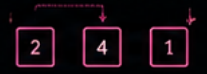
Output: `green`

choices(seq, k)

Return a list of k random elements from the population with replacement.

```
import random
print(random.choices([1,2,3,4], k=3))
```

Output: `[2, 4, 1]`

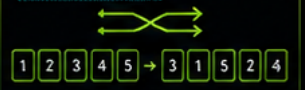


shuffle(seq)

Shuffle the sequence in place.

```
import random
lst = [1, 2, 3, 4, 5]
random.shuffle(lst)
print(lst)
```

Output: `[3, 1, 5, 2, 4]`



sample(pop, k)

Return a list of k unique elements chosen from the population.

```
import random
print(random.sample([10,20,30,40,50], 3))
```

Output: `[40, 10, 20]`

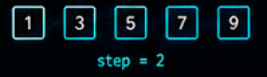
Unique Items (No Repetition)

randrange(start, stop, step)

Return a randomly selected element from range(start, stop, step).

```
import random
print(random.randrange(1, 10, 2))
```

Output: `7`



EXAMPLE: LOTTERY DRAW

```
import random
lucky_numbers = random.sample(range(1, 50), 6)
print("Lucky Numbers:", lucky_numbers)
```

7 12 23 31 42 48

USE CASES

- Games & Simulations
- Random Testing
- Lottery & Giveaways
- Data Sampling
- Shuffling Cards / Lists

PRO TIP

For reproducible results, set a seed!

```
import random
random.seed(42)
print(random.randint(1, 100))  # Always same output
```

IMPORTANT NOTES

- random() returns float in [0.0, 1.0).
- randint() is inclusive of both ends.
- shuffle() changes the original list.
- sample() returns unique items only.
- choices() allows duplicates.

MATH MODULE IN PYTHON

Mathematical Functions for Calculations & More

Built-in Module	Fast & Efficient	Wide Range	Accurate	Cross Platform
Always Available	Optimized for Performance	Trigonometry, Algebra & More	Reliable Results Every Time	Works on Any Python Platform

WHAT IS MATH MODULE?

The math module in Python provides access to mathematical functions and constants. It is part of Python Standard Library.

math

```
import math

# Use math functions
result = math.sqrt(25)
print(result)  # 5.0
```

No need to install anything. Just import and use!

MATH FUNCTIONS OVERVIEW

sqrt(x)	ceil(x)	floor(x)	pow(x, y)	factorial(n)	gcd(a, b)	pi	e
Square root of x	Smallest integer greater than x	Largest integer less than x	x raised to the power y	Factorial of n (n!)	Greatest Common Divisor of a, b	Value of Pi (π)	Value of Euler's number
sqrt(25) = 5.0	ceil(4.2) = 5	floor(4.8) = 4	pow(2, 3) = 8	factorial(5) = 120	gcd(48, 18) = 6	pi = 3.14159...	e = 2.71828...

All functions are in the math module. Import it once and use anywhere! math is always available in Python (no installation needed).

IMPORTANT CONSTANTS

math.pi
Value of Pi (π) 3.141592653589793

math.e
Value of Euler's Number (e) 2.718281828459045

Use these constants for accurate calculations.

EXAMPLES

```
import math

print("sqrt(16)    =", math.sqrt(16))
print("ceil(4.2)   =", math.ceil(4.2))
print("floor(4.8)  =", math.floor(4.8))
print("pow(2, 3)   =", math.pow(2, 3))
print("factorial(6)=", math.factorial(6))
print("gcd(48, 18) =", math.gcd(48, 18))
print("pi          =", math.pi)
print("e           =", math.e)
```

Output:

```
sqrt(16)     = 4.0
ceil(4.2)    = 5
floor(4.8)   = 4
pow(2, 3)    = 8.0
factorial(6) = 720
gcd(48, 18)  = 6
pi           = 3.141592653589793
e            = 2.718281828459045
```

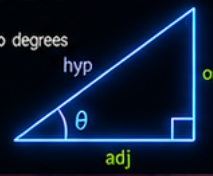
FUNCTION DETAILS

Function	Syntax	Description	Example	Result
sqrt(x)	math.sqrt(x)	Square root of x	math.sqrt(25)	5.0
ceil(x)	math.ceil(x)	Smallest integer ≥ x	math.ceil(4.2)	5
floor(x)	math.floor(x)	Largest integer ≤ x	math.floor(4.8)	4
pow(x, y)	math.pow(x, y)	x raised to power y	math.pow(2, 3)	8.0
factorial(n)	math.factorial(n)	Factorial of n (n!)	math.factorial(5)	120
gcd(a, b)	math.gcd(a, b)	Greatest common divisor	math.gcd(48, 18)	6
pi	math.pi	Value of Pi (π)	math.pi	3.141592653589793
e	math.e	Value of Euler's number	math.e	2.718281828459045

TRIGONOMETRIC FUNCTIONS (Examples)

- sin(x) Sine of x (x in radians)
- cos(x) Cosine of x (x in radians)
- tan(x) Tangent of x (x in radians)
- radians(x) Convert degrees to radians
- degrees(x) Convert radians to degrees

```
import math
print(math.sin(math.pi/2))  # 1.0
print(math.cos(0))          # 1.0
print(math.tan(math.pi/4))  # 1.0
print(math.radians(180))    # 3.14159...
print(math.degrees(math.pi)) 180.0
```



$\sin(x) = \frac{opp}{hyp}$

$\cos(x) = \frac{adj}{hyp}$

PRO TIP

Use math module for scientific calculations, competitive programming, data analysis & more!

BEST PRACTICES

- Always import math at the beginning.
- Use math functions for better accuracy.
- Use constants (pi, e) instead of hardcoding values.

USE CASES

- Scientific & Engineering Calculations
- Game Development (physics, angles)
- Data Science & Machine Learning
- Competitive Programming

NOTE

- All angles in trig functions are in radians.
- math.pow(x, y) returns float. Use ** for integers.
- factorial() works only for non-negative integers.

OS MODULE IN PYTHON

Interact with Operating System in Python

Built-in Module	Cross Platform	Powerful	Reliable	Versatile
Comes pre-installed with Python	Works on Windows, Linux, macOS	Perform OS related operations easily	Safe & efficient system access	File, Directory, Path, Env & more

WHAT IS OS MODULE?

The os module in Python provides a way to interact with the operating system. It allows us to perform operations like handling files, directories, environment variables and more.

```
import os
# OS operations made easy!
```

No installation needed. Just import and use!

COMMONLY USED FUNCTIONS

getcwd()	listdir()	mkdir(path)	makedirs(path)	remove(path)	rename(src, dst)	path	environment
Get current working directory	List files and directories in a path	Create a new directory	Create nested directories	Remove (delete) a file	Rename or move a file/directory	Work with file and directory paths	Access environment variables

The os module helps in automating system tasks and managing the file system efficiently.

QUICK EXAMPLE

```
import os

print("Current Directory:", os.getcwd())
print("Files & Folders:", os.listdir())
os.mkdir("TestDir")
print("Directory 'TestDir' created.")

os.rename("TestDir", "NewDir")
print("Renamed to 'NewDir'.")

os.remove("sample.txt")
print("File removed.")
```

FUNCTION EXAMPLES

getcwd()

```
import os
print(os.getcwd())

# Output
C:\Users\Lovejeet\Project
```

listdir()

```
import os
print(os.listdir())

# Output
['file1.txt', 'file2.py', 'Folder']
```

mkdir()

```
import os
os.mkdir("MyFolder")
print("Folder created")

# Creates 'MyFolder'
```

makedirs()

```
import os
os.makedirs("A/B/C")
print("Nested folders created")

# Creates A/B/C
```

PATH OPERATIONS (os.path)

os.path.join()	Join paths	os.path.join('A', 'B', 'C') # 'A/B/C'
os.path.exists()	Check if path exists	os.path.exists('file.txt') # True / False
os.path.isfile()	Check if path is file	os.path.isfile('file.txt') # True / False
os.path.isdir()	Check if path is directory	os.path.isdir('Folder') # True / False
os.path.basename()	Get file name from path	os.path.basename('/A/B/file.txt') # 'file.txt'
os.path.dirname()	Get directory name from path	os.path.dirname('/A/B/file.txt') # '/A/B'

ENVIRONMENT VARIABLES (os.environ)

```
import os
print(os.environ["USERNAME"])
print(os.environ["PATH"])

# Get a specific variable
print(os.getenv("HOME"))

# Set a new environment variable
os.environ["MY_VAR"] = "PythonRocks"
print(os.environ["MY_VAR"])
```

FILE OPERATIONS

remove() – Delete a file

```
import os
os.remove("file.txt")
print("File deleted")
```

rename() – Rename or Move

```
import os
os.rename("old.txt", "new.txt")
print("File renamed")
```

USE CASES

- Automate file & folder management
- Log file handling & cleanup
- System monitoring scripts
- Backup & restore operations
- Environment based configuration

PRO TIP

Use os.path.join() instead of hardcoding paths for cross-platform compatibility.

```
# Good Practice
file_path = os.path.join("Folder", "file.txt")
print(file_path)
```

NOTE

- Be careful with remove() – it permanently deletes files.
- Always check if file/directory exists before performing operations.
- Environment variables may differ across systems.

The os module is your bridge between Python and the Operating System.

Automate. Manage. Simplify. That's the power of os!

Import os and start exploring!

SYS MODULE IN PYTHON

Work with Python Interpreter & System

- **Built-in Module** — Comes pre-installed with Python
- **Access System** — Control the Python runtime environment
- **Powerful Tools** — Useful for scripts, debugging & more
- **Cross Platform** — Works on Windows, Linux, macOS
- **Essential Module** — Used by Python applications

WHAT IS SYS MODULE?

The sys module in Python provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter.

```
import sys
```

No installation needed.
Just import and use!

COMMONLY USED ATTRIBUTES & FUNCTIONS

- **argv** — Command line arguments list
- **path** — List of paths that Python searches
- **version** — Python version information
- **platform** — Platform information
- **exit()** — Exit the Python interpreter
- **stdin** — Standard input stream
- **stdout** — Standard output stream

The sys module gives you control over the Python interpreter and its environment.

QUICK EXAMPLE

```
import sys

print("Python Version:", sys.version)
print("Platform:"; sys.platform)
print("Command Line Arguments:", sys.argv)
print("Python Path:")
for p in sys.path:
   print(" -", p)
print("Standard Input:", sys.stdin)
print("Standard Output:", sys.stdout)

# Exit the program
# sys.exit("Goodbye!")
```

IMPORTANT ATTRIBUTES

Attribute	Description	Example
sys.argv	List of command line arguments passed to the script	sys.argv[0] # Script name sys.argv[1:] # Arguments
sys.path	List of directories that Python searches for modules	for p in sys.path: print(p)
sys.version	Detailed Python version information	sys.version
sys.platform	Platform identifier	sys.platform
sys.stdin	Standard input stream	sys.stdin.read()
sys.stdout	Standard output stream	sys.stdout.write("Hello")
sys.stderr	Standard error stream	sys.stdout.write("Error!")

IMPORTANT FUNCTIONS

Function	Description	Example
sys.exit([code])	Exit program with optional exit code (default 0)	sys.exit(0) sys.exit("Done")
sys.getsizeof(obj)	Return size of an object in bytes	sys.getsizeof(123)
sys.getrecursionlimit()	Return current recursion limit	sys.getrecursionlimit()
sys.setrecursionlimit(n)	Set recursion limit	sys.setrecursionlimit(1000)
sys.maxsize	Maximum value a Python int can take	sys.maxsize

sys.argv EXAMPLE

Script: myscript.py

```
import sys
print("Arguments:"; sys.argv)
```

Terminal:

```
python myscript.py hello 123 test
```

Output:

```
Arguments: ['myscript.py', 'hello', '123', 'test']
```

STDIN & STDOUT EXAMPLE

Read from user (stdin):

```
import sys
name = sys.stdin.readline().strip()
print("Hello,", name, file=sys.stdout)
```

stdin → stdout

USE CASES

- Command line argument parsing
- Cross-platform compatibility
- Logging & debugging
- Environment information
- Control Python interpreter

PRO TIP

- Use sys.exit() to exit gracefully.
- Use sys.path to add custom module paths.
- Use sys.argv for building command line tools.

NOTE

- sys.exit() stops the interpreter immediately.
- Be careful while modifying sys.path.
- stdout and stderr are file-like objects.
- Changes in sys affect the interpreter.

sys module – Your Window to Python Interpreter! | Understand. Control. Automate. | import sys and explore the power!

COLLECTIONS MODULE IN PYTHON

Specialized Container Datatypes for Efficient Data Handling

- Optimized Data Structures for Real-World Problems
- Performance Better Than Built-in Containers
- Made for Counting, Grouping & Queue Operations
- Easy to Use & Highly Flexible
- Part of Python Standard Library

WHAT IS COLLECTIONS MODULE?

The collections module provides specialized container datatypes beyond Python's built-in datatypes (list, dict, set, tuple). These containers are designed for specific use cases and offer better performance and convenience.

```
from collections import *
```

Import only what you need or import the specific class to keep your code clean!

OVERVIEW OF COLLECTIONS DATA STRUCTURES

Counter
A dict subclass for counting hashable objects.

Use Cases:
- Counting elements
- Frequency analysis
- Most common items

defaultdict
A dict subclass that provides a default value for missing keys.

Use Cases:
- Grouping data
- Avoid KeyError
- Nested dictionaries

deque
A list-like container with fast appends and pops from both ends.

Use Cases:
- Queue / Stack
- Sliding window
- Undo operations

namedtuple
Creates tuple subclasses with named fields for better readability.

Use Cases:
- Structured data
- Clean attribute access
- Immutable records

ChainMap
A dict-like class for creating a single view of multiple mappings.

Use Cases:
- Layered configuration
- Scope management
- Merging dictionaries

OrderedDict
A dict subclass that remembers the order of keys (pre Python 3.7).

Use Cases:
- Maintains order
- Ordered iteration
- Move & reordering

WHY USE COLLECTIONS?

- More specialized & efficient than built-in types
- Cleaner and more readable code
- Built for common real-world patterns
- Reduces boilerplate code
- Improves performance in many scenarios

```
from collections import Counter
words = ["apple", "banana", "apple", "orange", "banana", "apple"]
count = Counter(words)
print(count)  # Counter({'apple': 3, 'banana': 2, 'orange': 1})
```

Collections module = Power Tools for Python Developers!

DETAILED GUIDE WITH EXAMPLES

1. Counter
Count hashable objects.

```
from collections import Counter
nums = [1, 2, 2, 3 3, 3, 4]
c = Counter(nums)
print(c)
# Output: Counter({3: 3, 2: 2, 1: 1, 4: 1})
print(c.most_common(2))  # [(3, 3), (2, 2)]
print(c[3])  # 3
```

Common Methods

most_common(n)	Return n most common elements
elements()	Expand counts to elements
update(iterable)	Update counts from iterable
subtract(iterable)	Subtract counts from iterable

2. defaultdict
Dictionary with default value for missing keys.

```
from collections import defaultdict
d = defaultdict(int)
d['a'] += 1
d['a'] += 2
d['b'] += 5
print(dict(d))  # {'a': 3, 'b': 5}
```

Common Use

Grouping items in a list:

```
d = defaultdict(list)
d['fruits'].append('apple')
d['fruits'].append('banana')
print(dict(d))  # {'fruits': ['apple', 'banana']}
```

3. deque
Double-ended queue with fast operations.

```
from collections import deque
d = deque([1, 2, 3])
d.append(4)       # add right
d.appnedleft(0)   # add left
print(d)      # deque([0, 1, 2, 3, 4])
d.pop()       # remove right  -> 4
d.popleft(1)  # remove left   -> 0
```

Common Methods

append(x)	Add item to right
appendleft(x)	Add item to left
pop() / popleft()	Remove from right/left
rotate(n)	Rotate n steps

4. namedtuple
Tuple subclass with named fields.

```
from collections import namedtuple
Person = namedtuple('Person', ['name', 'age'])
p = Person('Alice', 25)
print(p.name)  # Alice
print(p.age)   # 25
print(p)       # Person(name='Alice', age=25)
```

Advantages

- Readable & self-documenting
- Access using dot notation
- Immutable (like tuple)

5. ChainMap
Combine multiple dictionaries into one view.

```
from collections import ChainMap
a = {'x': 1, 'y': 2}
b = {'y': 20, 'z': 30}
chain = ChainMap(a, b)
print(chain['x'])   # 1 (from a)
print(chain['y'])   # 2 (from a, first map)
print(chain['z'])   # 30 (from b)
print(dict(chain)) # {'y': 2, 'z': 30, 'x': 1}
```

Common Methods

maps	List of underlying mappings
new_child(m)	Add new child mapping
parents	Return new ChainMap without first

6. OrderedDict
Dictionary that remembers insertion order.

```
from collections import OrderedDict
od = OrderedDict()
od['a'] = 1
od['b'] = 2
od['c'] = 3
print(od) # OrderedDict([('a', 1), ('b', 2), ('c', 3)])
od.move_to_end('a')
print(od) # OrderedDict([('b', 2), ('c', 3), ('a', 1)])
```

Common Methods

move_to_end(key)	Move key to end
popitem(last=True)	Pop last/first item
keys() / values()	Return keys/values in order

COLLECTIONS MODULE SUMMARY

Class	Type	Best For
Counter	dict subclass	Counting & frequency
defaultdict	dict subclass	Default values & grouping
deque	sequence	Queue, stack, fast operations
namedtuple	tuple subclass	Structured & immutable data
ChainMap	dict-like	Combine multiple mappings
OrderedDict	dict subclass	Remember key order

USE CASES AT A GLANCE

- Count occurrences of items → Counter
- Group items by key → defaultdict(list)
- Implement queue/stack → deque
- Store records with names → namedtuple
- Merge configurations → ChainMap
- Maintain order while working → OrderedDict

PRO TIP

Use the right tool for the job:

- Performance matters
- Clean & readable code
- Collections makes it EASY!

BEST PRACTICES

- Use Counter for counting, not dictionary
- Use defaultdict to avoid KeyError
- Use deque for queues & sliding windows
- Use namedtuple for clean, readable code
- Use ChainMap for layered configs
- Use OrderedDict only when order matters

NOTE

- From Python 3.7+, dict remembers insertion order by default, but OrderedDict still gives extra methods.
- All collections are in the collections module (standard library).

ITERTOOLS MODULE IN PYTHON

Powerful Tools for Efficient Iteration and Data Processing

- **High Performance** — Optimized for speed
- **Memory Efficient** — Iterators create values on demand
- **Easy to Use** — Simple functions with big power
- **Versatile** — Works with any iterable
- **Built-in Module** — Part of Python Standard Library

WHAT IS ITERTOOLS?

The **itertools** module provides a collection of functions that create **iterators** for efficient looping and combinatorics. These functions are memory efficient and fast.

```
import itertools
# Import what you need
```

Itertools returns iterators, not lists. Use with for loop or convert if needed.

MOST COMMONLY USED FUNCTIONS

count()
Return evenly spaced values starting with start (default 0).

```
from itertools import count
for i in count(5):
    if i > 8: break
    print(i, end=' ')
```

Output:
5 6 7 8

cycle()
Return elements from the iterable, cycling through them forever.

```
from itertools import cycle
c = cycle('ABCD')
for _ in range(8):
    print(next(c), end=' ')
```

Output:
A B C D A B C D

repeat()
Repeat the value specified number of times or forever.

```
from itertools import repeat
for x in repeat('Hi', 3):
    print(x, end=' ')
```

Output:
Hi Hi Hi

chain()
Combine multiple iterables into a single iterable sequentially.

```
from itertools import chain
for x in chain('AB', [1, 2], 'CD'):
    print(x, end=' ')
```

Output:
A B 1 2 C D

product()
Cartesian product of input iterables. Equivalent to nested loops.

```
from itertools import product
for p in product('AB', '12'):
    print(p, end=' ')
```

Output:
('A', '1') ('A', '2')
('B', '1') ('B', '2')

permutations()
All possible orderings (r-length permutations) of elements.

```
from itertools import permutations
for p in permutations('ABC', 2):
    print(p, end=' ')
```

Output:
('A', 'B') ('A', 'C') ('B', 'A')
('B', 'C') ('C', 'A') ('C', 'B')

C(n,r) combinations()
All possible combinations (r-length) of elements without repetition.

```
from itertools import combinations
for p in combinations('ABC', 2):
    print(c, end=' ')
```

Output:
('A', 'B') ('A', 'C') ('B', 'C')

MORE USEFUL ITERTOOLS FUNCTIONS

- **CWR** combinations_with_replacement() — Combinations with replacement allowed.
- **Σ** accumulate() — Cumulative sums or custom binary function.
- **G** groupby() — Group consecutive elements by key function.
- islice() — Extract slices from an iterable.
- starmap() — Map function arguments from iterable.
- tee() — Split one iterator into n independent iterators.

QUICK EXAMPLES

combinations_with_replacement()

```
from itertools import
combinations_with_replacement
for c in combinations_with_replacement('AB',2):
    print(i, end=' ')
```

Output:
('A', 'A') ('A', 'B') ('B', 'B')

accumulate()

```
from itertools import accumulate
from operator import add
print(list(accumulate([1, 2, 3, 4], add)))
```

Output:
[1, 3, 6, 10]

groupby()

```
from itertools import groupby
data = 'AAABBBCCDAA'
for k, g in groupby(data):
    print(k, list(g))
```

Output:
A ['A', 'A', 'A']
B ['B', 'B', 'B']
C ['C', 'C']
D ['D']
A ['A', 'A']

islice()

```
from itertools import islice
print(list(islice(range(10), 2, 8, 2)))
    # start=2, stop=8, step=2
```

Output:
[2, 4, 6]

starmap()

```
from itertools import starmap
from operator import add
pairs = [(1,2), (3,4), (5,6)]
print(list(starmap(add, pairs)))
```

Output:
[3, 7, 11]

tee()

```
from itertools import tee
it = iter('ABCDE')
a, b = tee(it)
print(list(a))  # ['A', 'B', 'C', 'D', 'E']
print(list(b))  # ['A', 'B', 'C', 'D', 'E']
```

USE CASES

- Looping infinite sequences
- Combinatorial problems
- Data analysis & grouping
- Generating test data
- Efficient iterators for large datasets

PRO TIP

Itertools returns iterators, not lists.
Use in for loop directly to save memory.

```
from itertools import count
for i in count():  # infinite!
    if i > 1000: break
    # process i
```

BEST PRACTICES

- Use itertools for memory efficiency
- Prefer iterators over lists
- Chain iterables instead of nested loops
- Convert to list only when necessary
- Combine with other modules (functools, operator)

NOTE

- These functions return iterators
- Use next() or for loop to get values
- Some iterators are infinite (count, cycle, repeat)
- Convert to list carefully (may use more memory)

itertools – Make your iteration smarter, faster and more Pythonic! | Built-in • Fast • Efficient • Pythonic | from itertools import * | Happy Coding!

PYTHON QUICK REFERENCE

Essential Python Syntax, Concepts & Examples at a Glance

Easy to Learn — Simple & Readable
Versatile — Web, Data, AI & more
Powerful — Built-in tools get the job done
Reliable — Robust & Production Ready
Huge Community — Support & Resources

1. VARIABLES

- No need to declare type
- Use meaningful names
- Case-sensitive

Examples

```
x = 10
name = "CodeSardar"
price = 19.99
is_valid = True
```

2. DATA TYPES

Numeric

```
int       x = 10
float     y = 3.14
complex   z = 2 + 3j
```

Sequence

```
str     s = "Hello"
list    lst = [1, 2, 3]
tuple   t = (1, 2, 3)
```

Other

```
bool      b = True
set       st = {1, 2, 3}
dict      d = {"a": 1, "b": 2}
NoneType  n = None
```

3. FUNCTIONS

- Use def to define a function
- Can return values
- Supports default & keyword args

```
def greet(name, msg="Hi"):
    return f"{msg}, {name}!"

print(greet("Jeet"))
# Hi, Jeet!
```

Lambda (anonymous function)

```
add = lambda a, b: a + b
print(add(4, 5))  # 9
```

4. LISTS

- Ordered collection
- Mutable (changeable)
- Supports duplicates

Examples

```
nums = [1, 2, 3, 4]
nums.append(5)      # [1, 2, 3, 4, 5]
nums[0]             # 1
nums[-1]            # 5
nums[1:3]           # [2, 3]
for n in nums:
    print(n)
# 1 2 3 4 5
```

5. DICTIONARIES

- Key-value pairs
- Keys must be unique
- Mupable and unordered

Examples

```
student = {"name": "Jeet", "age": 32}
student["city"] = "Amritsar"
student["name"]     # Jeet
student.get("age")  # 32
for k, v in student.items():
    print(k, "=", v)
# name = Jeet
# age = 32
# city = Amritsar
```

6. OOP (CLASSES)

- Class = Blueprint
- Object = Instance
- Supports Inheritance, Polymorphism

```
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def info(self):
        return f"{self.name}, {self.age}"

s1 = Student("Jeet", 32)
print(s1.info())  # Jeet, 32
```

7. MODULES

- Reusable Python files
- Use import to include
- Built-in & custom modules

Examples

```
import math
print(math.sqrt(16))      # 4.0

from datetime import datetime
now = datetime.now()
print(now.strftime("%Y-%m-%d"))
```

Import Types

- import module
- from module import name
- from module import *

8. EXCEPTIONS

- Handle errors gracefully
- Use try, except, else, finally
- Keeps program from crashing

```
try:
    x = int(input("Enter a number: "))
    print(10 / x)
except ZeroDivisionError:
    print("Cannot divide by zero!")
except ValueError:
    print("Invalid input!")
else:
    print("Success!")
finally:
    print("Done!")
```

9. POPULAR LIBRARIES

NumPy	Numerical computing with arrays & matrices
Pandas	Data analysis and manipulation
Matplotlib	Data visualization and plotting
Requests	HTTP requests made easy
BeautifulSoup	Web scraping made simple

EXTRA QUICK SYNTAX

Input	`name = input("Your name: ")`
Output	`print("Hello, World!")`
If-Else	`if x > 0: print("Positive") else: print("Non-positive")`
Loops	`for i in range(5): print(i)` `while x < 10: x += 1`
List Comp.	`squares = [x*x for x in range(5)]`
String Format	`f"Name: {name}, Age: {age}"`
None Check	`if value is None: ...`

HANDY OPERATORS

Arithmetic	`+ - * / // % **`
Comparison	`== != > < >= <=`
Logical	`and or not`
Bitwise	`& | ^ ~ << >>`
Membership	`in not in`
Identity	`is is not`
Assignment	`= += -= *= /== %= **=`

PRO TIP

Practice regularly. Build projects. Read docs. Learn by doing!

PYTHONIC STYLE

- Readable Code is Better Code
- Follow PEP 8 Style Guide
- Keep it Simple & Pythonic

REMEMBER

- Indentation matters
- Everything is an object
- Use help() to learn more

USEFUL BUILT-IN FUNCTIONS

`len()` `type()` `range()` `print()` `input()`
`sum()` `max()` `min()` `sorted()` `open()`

Keep Coding, Keep Growing!